

Visual C++ 图形用户界面 开发指南

李博轩 等 编著

清华大学出版社

(京)新登字 158 号

内 容 简 介

本书通过大量实例深入浅出地介绍了 Visual C++ 图形用户界面开发技术,对 Windows 界面中最重要的组成元素分别进行了介绍。全书共 10 章,主要内容包括:Windows 用户界面制作基础、按钮控件、编辑控件、组合框控件、列表视图控件、树视图控件、菜单、工具栏、状态栏、框架窗口等编程技术。对每种编程技术,都给出了具有代表性的应用实例,使读者能够通过实例的学习,迅速掌握图形用户界面编程技术。

本书内容全面、深入,适合中高级读者、大专院校师生、企业技术人员学习参考,也适合各类高级培训班学员学习 Visual C++ 图形用户界面开发技术。

版权所有,翻印必究。

本书封面贴有清华大学出版社激光防伪标签,无标签者不得销售。

书 名: Visual C++ 图形用户界面开发指南

作 者: 李博轩 等

出版者: 清华大学出版社(北京清华大学校内,邮编 100084)

<http://www.tup.tsinghua.edu.cn>

印刷者: 北京市清华园胶印厂

发行者: 新华书店总店北京发行所

开 本: 787×1092 1/16 **印张:** 26.25 **字数:** 621 千字

版 次: 2000 年 11 月第 1 版 2000 年 11 月第 1 次印刷

书 号: ISBN 7-900630-68-6

印 数: 0001 ~ 5000

定 价: 46.80 元

前 言

我们设计的程序是否能够吸引用户,是否总能寄希望于程序精彩的内涵呢?“酒香不怕巷子深”并不是在任何情况下都适用。在一个软、硬件技术不断发展的时代,对大多数并非计算机专家的用户,在他们面临多种选择时,怎样才能使自己的产品脱颖而出?这对大多数程序,尤其是商用程序来说是一个相当重要的问题。

显然,别具一格的程序外观足以影响用户的选择。拥有友好而富于个性的用户界面的软件,往往具有更强的功能。这也是经过事实证明了。

另一方面,随着开发工具的不断发展,程序员所要做的工作越来越少,而这必然导致系统的许多细节被隐藏起来,能供程序员自由发挥的空间越来越小。其最直观的表现就是,通过 Visual C++ 开发的应用程序具有极其类似的界面。如果你希望编写出富有个性的界面,就必须付出更多的精力。

漂亮的界面有利而无弊,可是我们怎么实现它呢?

本书对 Windows 界面中最重要的组成元素分别进行了介绍。对于每个元素,都通过大量经典实例深入浅出地向读者展示:如何使其具有更强的功能和更漂亮的外观。但是本书绝非代码的堆积,而是试图通过对经典实例的分析,使读者理解界面设计的原则和方法,并进而对 Windows 高级编程技术有更深入的理解。

本书侧重 Visual C++ 在图形用户界面方面的应用,重点介绍如何利用 Visual C++ 实现应用系统的图形用户界面开发。全书按应用程序界面开发的不同主题来编排内容,分别讲述 Windows 用户界面制作基础、按钮控件、编辑控件、组合框控件、列表视图控件、树视图控件、菜单、工具栏、状态栏、框架窗口等。对于每个主题,都给出了 Visual C++ 在这一方面的要领,以及运用这类技术的实例和技巧,使读者能够通过实例的学习,迅速掌握图形用户界面编程技术。

本书所附的光盘中含有全部实例的源代码。

本书的特点在于讲述如何用 Visual C++ 实现应用程序的图形用户界面,而且将 Visual C++ 的图形用户界面特性以及技术难点融入到具体的实例中。不强调“大而全”,而侧重“专而精”。

参与本书编写的除封面署名作者外,还有吴灵、刘秀蓉、吉尚戎、吉二源、杜丽、何震声、宋森、陈明、李洪声、刘海涛、李敏、刘志诚、朱志言、刘兵,另外,王宇红、李兵、刘海兰、姚文龙、李晓霞、向文兵、刘斌、张勇、张碧霞、孟文征参与了校对与录排工作,在此对他们的辛勤劳动表示感谢。由于水平所限,不足之处恳请读者批评指正。

作 者

2000 年 7 月

目 录

第 1 章 Windows 用户界面制作基础	1
1.1 用户界面设计基础	1
1.1.1 设计原则	2
1.1.2 Windows 界面规则	3
1.1.3 界面布局原则	4
1.1.4 用户辅助模型	6
1.2 Windows 编程机制	7
1.2.1 消息驱动	7
1.2.2 MFC 类库	10
1.2.3 MFC 框架与消息处理	16
1.3 Windows 应用程序结构体系	17
1.3.1 文档/视图结构概述	17
1.3.2 文档和 CDocument 类	18
1.3.3 视图与 CView 类	20
1.3.4 框架窗口	22
本章小结	22
第 2 章 按钮控件	23
2.1 按钮控件编程基础	23
2.1.1 按钮控件概述	23
2.1.2 创建函数	24
2.1.3 操作函数	26
2.1.4 重载函数	29
2.1.5 CBitmap Button 类	31
2.2 改变按钮颜色	33
2.2.1 设计彩色按钮管理类	33
2.2.2 使用彩色按钮管理类	38
2.3 改变按钮形状	39
2.3.1 创建多边形按钮	39
2.3.2 创建圆形按钮	43
2.3.3 创建球形按钮	45
2.4 动态创建高级按钮	51

2.4.1	设计高级按钮管理类	51
2.4.2	动态创建	53
2.4.3	按钮绘制过程分析	54
2.4.4	使用高级按钮管理类	57
	本章小结	60
第3章	编辑控件	61
3.1	编辑控件编程基础	61
3.1.1	编辑控件概述	61
3.1.2	构造函数	62
3.1.3	属性操作函数	64
3.1.4	常规操作函数	68
3.1.5	剪贴板操作	72
3.2	CEditView 类	73
3.2.1	CEditView 类概述	74
3.2.2	构造函数	74
3.2.3	属性操作函数	74
3.2.4	常规操作函数	76
3.2.5	重载函数	77
3.3	改变控件的外观	78
3.3.1	能够保持“高亮”状态的编辑控件	78
3.3.2	鼠标敏感编辑控件	80
3.4	改变控件的编辑及显示方式	83
3.4.1	限制输入的数据类型	83
3.4.2	在位编辑	95
3.4.3	语法着色	98
	本章小结	100
第4章	组合框控件	101
4.1	组合框控件编程基础	101
4.1.1	组合框控件概述	101
4.1.2	构造函数	103
4.1.3	常规操作函数	105
4.1.4	字符串操作函数	112
4.1.5	重载函数	115
4.2	改变组合框控件的行为	118
4.2.1	自动完成组合框控件	118
4.2.2	使用工具窗口替代列表框	124
4.2.3	鼠标敏感组合框控件	125

4.3	改变组合框控件选项形式	129
4.3.1	图标选择组合框控件	129
4.3.2	字体选择组合框控件	133
4.3.3	颜色选择组合框	146
4.4	增强列表框控件	149
4.4.1	CComboBoxEx 类概述	150
4.4.2	常用操作编程	153
	本章小结	156
第 5 章	列表视图控件	157
5.1	列表视图控件编程基础	157
5.1.1	构造函数	157
5.1.2	属性操作函数	158
5.1.3	常规操作函数	176
5.1.4	虚函数	182
5.2	列表视图控件常用操作编程	182
5.2.1	创建列表视图控件	182
5.2.2	向控件中添加新条目和新列	183
5.2.3	改变控件的扩展风格	184
5.2.4	使用图像列表	184
5.2.5	操作控件的工作区域	185
5.2.6	虚列表控件	186
5.3	一个经典话题	188
5.4	动态改变列表视图的行高	196
5.5	改变列表视图控件的背景	199
5.5.1	改变背景颜色	199
5.5.2	使用位图背景	201
5.6	改善列表视图控件的交互方式	207
5.6.1	在列表视图控件中使用复选框	207
5.6.2	在位编辑子项	208
5.6.3	使用组合框控件	216
5.6.4	增强子项在位编辑性能	223
5.6.5	内容提示	231
5.6.6	改进内容提示	238
5.7	改变列表视图控件的标头显示	240
5.7.1	在标头中显示图像	240
5.7.2	在标头中使用图像列表	241
	本章小结	245

第 6 章 树视图控件	246
6.1 树视图控件基础	246
6.1.1 树视图控件概述	246
6.1.2 构造函数	247
6.1.3 属性操作函数	248
6.1.4 常规操作函数	261
6.2 条目基本操作编程	267
6.2.1 展开分支	267
6.2.2 收拢分支	268
6.2.3 收拢所有分支	268
6.2.4 拷贝条目	269
6.2.5 拷贝分支	270
6.2.6 移动条目或分支	270
6.2.7 得到分支中的最后一个条目	270
6.2.8 得到控件中的下一个条目	271
6.2.9 得到控件中的上一个条目	272
6.3 条目图像编程	272
6.3.1 设置条目图像	273
6.3.2 设置状态图像	274
6.3.3 使用覆盖图像	274
6.4 条目检索操作编程	275
6.4.1 检索匹配标签	275
6.4.2 检索匹配数据	277
6.4.3 检索匹配 TV_ITEM 结构	278
6.5 编辑条目标签	280
6.5.1 编辑标签	280
6.5.2 使用 Esc 和 Return 键结束编辑	281
6.5.3 禁止编辑标签	282
6.5.4 树视图控件状态	282
6.6 树视图控件的拖拽操作	283
6.6.1 实现拖拽	284
6.6.2 处理无意拖拽	286
6.6.3 使用 Esc 取消拖拽	287
6.6.4 处理拖拽操作中的滚动问题	288
6.6.5 在拖拽中保持条目等级	290
6.6.6 增强拖拽功能	291
6.7 树视图控件与工具提示	293
6.7.1 为条目图像添加工具提示	294

6.7.2 为条目添加工具提示	297
6.8 实现多重选择	298
6.9 改善条目形式和外观	303
6.9.1 鼠标敏感条目	303
6.9.2 为条目添加复选框	305
6.9.3 改变条目的字体和颜色	310
6.10 改善控件外观	314
6.10.1 改变控件背景颜色	315
6.10.2 使用位图背景	317
6.11 序列化树视图控件内容	321
6.12 目录浏览器	322
本章小结	323
第7章 菜单	324
7.1 菜单编程基础	324
7.1.1 构造函数	324
7.1.2 初始化函数	325
7.1.3 菜单操作函数	328
7.1.4 菜单项操作函数	329
7.1.5 重载函数	338
7.2 使用标准菜单	339
7.3 使用快捷菜单	341
7.4 使用动态菜单	341
7.4.1 动态创建/修改菜单	341
7.5 使用自绘制菜单	343
7.5.1 彩色菜单	343
7.5.2 图标菜单	347
本章小结	351
第8章 工具栏	352
8.1 工具栏编程基础	352
8.1.1 工具栏概述	352
8.1.2 构造函数	353
8.1.3 属性操作函数	356
8.2 使用标准工具栏	359
8.3 创建 IE 风格的工具栏	360
8.3.1 使工具栏具有“热敏”变色风格	360
8.3.2 在工具栏中显示文本	361
8.4 创建下拉菜单式工具栏按钮	362
8.5 在工具栏中使用控件	365

8.5.1 添加组合框控件	365
8.5.2 添加复选框控件	366
8.6 使用 16M 色位图创建工具栏	369
8.7 去除浮动工具栏中的系统菜单	369
8.8 排列多个工具栏	370
8.9 在对话框中使用工具栏和工具提示	371
8.9.1 创建工具栏	372
8.9.2 修改对话框尺寸	372
8.9.3 显示工具提示	373
8.10 在 MDI 应用程序中切换工具栏	375
本章小结	378
第 9 章 状态栏	379
9.1 状态栏编程基础	379
9.1.1 状态栏概述	379
9.1.2 构造函数	380
9.1.3 属性操作函数	381
9.1.4 重载函数	384
9.2 使用标准状态栏	384
9.3 在状态栏中显示滚动效果的文本	385
9.4 在状态栏中输出时间	387
9.5 动态改变状态栏中的默认提示	389
9.6 在状态栏中使用控件	391
9.6.1 设计通用控件状态栏类	391
9.6.2 设计控件友元类	393
9.6.3 应用实例	397
9.6.4 使用自定义消息响应状态栏控件动作	398
9.6.5 使用注册窗口消息响应状态栏控件动作	399
本章小结	400
第 10 章 框架窗口	401
10.1 改变窗口效果	401
10.1.1 应用程序的默认图标	401
10.1.2 修改窗口的默认风格	402
10.1.3 改变窗口标题	404
10.1.4 改变窗口位置和排列	405
10.1.5 改变窗口形状	406
10.2 添加闪屏效果	407
10.3 添加窗口背景	408
本章小结	410

第 1 章 Windows 用户界面制作基础

除非应用程序只是给自己使用,否则它的价值必须由别人决定。应用程序的界面对用户的影响很大——无论程序代码如何高效,功能如何强大,如果用户发现它太难于使用,那么这个程序就不会得到广泛的应用。

本章主要向读者介绍应用程序界面设计原则、Windows 编程机制以及 Visual C++ 应用程序框架的结构和功能。由于现在 Visual C++ 为用户所做的工作越来越多,造成用户对 Windows 编程的机制的了解反而愈来愈少。只有充分理解 Windows 应用程序的结构和运行机制,才能真正掌握 Windows 编程。本书主要关注应用程序界面的设计,而如果希望设计出独特而有创意的界面,那么就必须对 Windows 内部机制有一个较为深入的理解。

本章要点:

- 界面设计基础;
- Windows 编程机制;
- Windows 应用程序结构体系。

1.1 用户界面设计基础

并不是只有拥有出色的艺术天赋,才能创建出优秀的用户界面——绝大多数界面设计规则与小学图画课上讲的完全一致。在那些课上介绍的组分、颜色的设计规则同样也适用于计算机屏幕,这与使用蜡笔在纸上画图没有什么不同。

对于使用可视工具编程的程序员来说,创建用户界面是一件非常容易的事情,简单到只需将控件拖动到对话框中,或在代码中调用一两个函数将自己绘制的位图设置为工具栏。然而正是这种简单性,使得创建出的应用程序越来越千篇一律,缺乏个性。关于这一点,可能读者也有体会。在用户界面设计上多花一些心思不仅能美观应用程序,提高易用性,而且能够增加程序对用户的吸引力(当然,从根本上说程序的功能才是最重要的因素)。

在设计用户界面之前,最好首先参考一下那些销路很好的软件界面,如果可能再参考一些销路并不很好的软件界面,这样可以得到很多有用的信息。而且自己在使用软件的过程中,对于用户界面也可能有一些想法和期望。例如对于某个软件,你可能对它的一些地方很满意而对另一些则不太满意,那么就可以在自己的设计中加以借鉴和改进。当然,个人的看法可能并不太适合大多数用户,因此也需要用户的确认。

成功的应用程序都会为用户提供多样化的选择。例如,Microsoft Internet Explorer 允许用户使用菜单命令、键盘命令或直接拖放来进行文件拷贝。提供多样化选择,能够提高应用程序的吸引力。在应用程序中同一功能,应该至少能够通过键盘或鼠标完成。

1.1.1 设计原则

作为程序员,当然应该十分精通计算机,然而大多数用户可能并不太理解(或不关心)应用程序的技术问题。他们只是将应用程序作为一种更简单的解决问题的工具。一个好的应用程序界面应当将底层技术与用户分开,就像 MFC 类一样将底层功能包装在一个个的函数中。

用户界面设计的一个总的原则就是用户至上。例如应用程序的设计是否能够吸引用户;在没有帮助的情况下,用户是否能够比较容易地发现应用程序的不同功能;应该如何为用户提供在线帮助等等。所有的这些都要要求程序员考虑问题时,必须更多地从用户的角度出发。

在实际使用中,软件和硬件并不能实现无缝结合,硬件的发展速度和软件的更新并不同步。这也就是说,错误几乎是不可避免的,因此错误处理也是用户界面设计的一部分。当出现错误时,通常的方法是显示一个对话框,并询问用户应该如何处理。而另一种不太常用的方法(但是更值得使用)就是自己后台解决,而不是麻烦用户。总而言之,用户关心的是完成特定的任务,而不是技术细节。因此在设计用户界面时,需要考虑到可能出现的错误,并确定哪些需要与用户交互处理,而哪些可以自己内部解决。

当应用程序出现错误时,如果需要与用户进行交互,一般是通过对话框完成,而在代码中则使用分支语句(例如 if...else 语句)。对话框是应用程序界面的一部分,它的设计对于应用程序的易用性也非常重要。

有一个笑话:在应用程序中设计很多对话框的程序员,在自己的人际交往中往往不能与他人很好地交流。诸如“A sector of fixed disk C: is corrupted or inaccessible. Abort, Retry, Ignore?”(如图 1-1)的消息对于大多数用户来说毫无意义。可以设想,服务员问您“可能饭菜卖完了或者厨房失火了。取消、重试还是忽略?”,这时您将如何回答?实际上,这告诉我们应该以用户能够理解的方式来提示选择,例如图中下方的对话框中的消息“*There is a problem saving your file on drive C. Save file on drive A, Don't save the file?*”就友好得多。

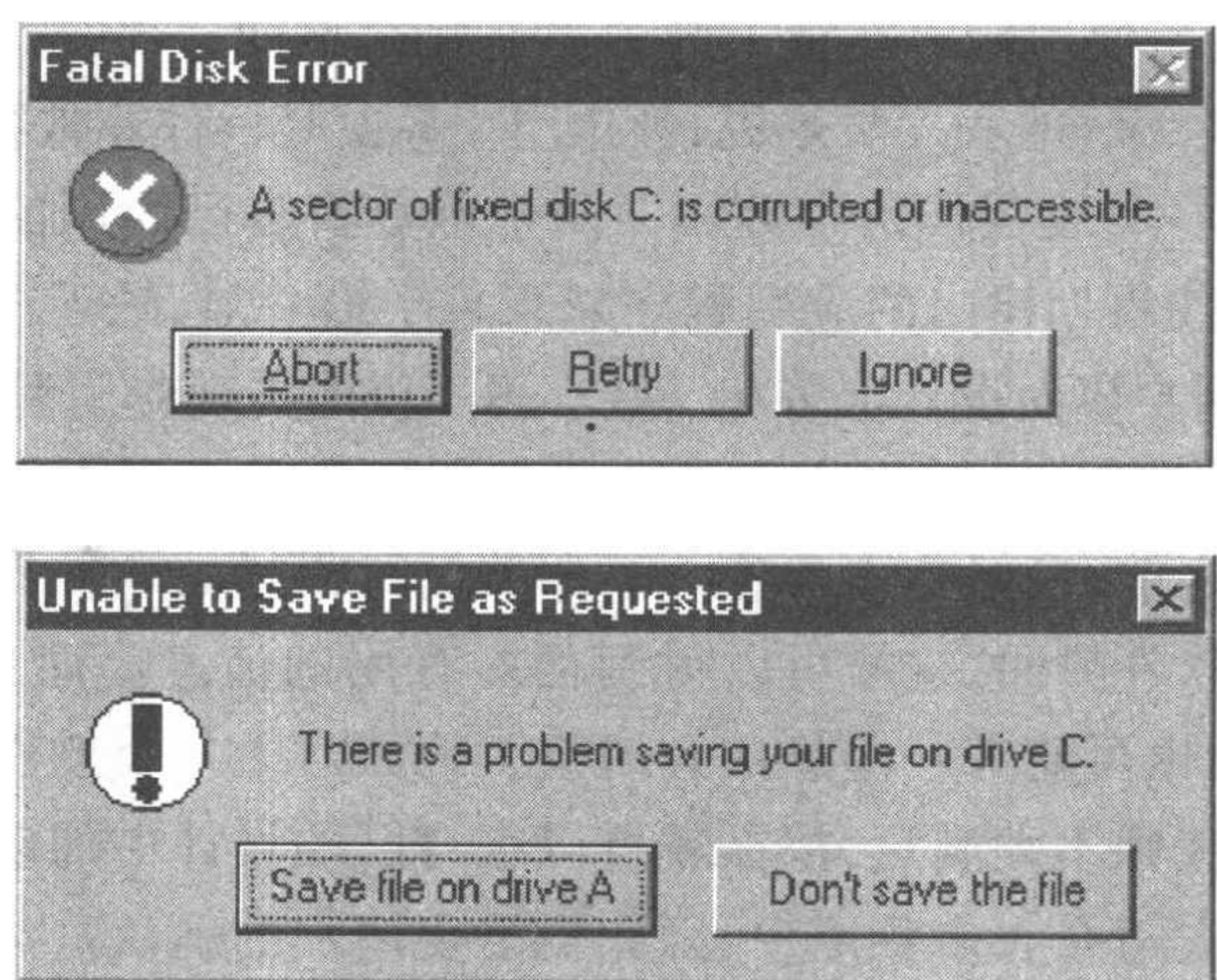


图 1-1 使用更易于理解的语言

在应用程序中设计错误信息对话框时,首先考虑这个信息对于用户是否有用;其次考

虑按钮所给出的选项意思是否清楚、选项和消息是否适合等。此外,如果仅仅显示一个错误信息会给用户造成不好的印象。关于这一点,可能读者也有体会,例如 Windows 著名的蓝屏现象。

并不是所有的错误都需要用户进行处理。对于在代码中就能够处理的错误,就无需通知用户,或者以不中断当前进度的方式进行通知。关于这一点,Word 里面的“自动更正”功能就是一个很好的例子:如果常用词出现错误,则自动修改;而如果不常用的词出现错误,则以红色波浪线表示它,以使用户以后修改。错误处理的技术有好几种,但哪个更适合就需要程序员自己作出决定。下面是一些通用的规则:

- 在“编辑”菜单中添加“撤消”命令,这要比打断用户当前的操作,并显示对话框询问是否要撤消好一些。
- 在状态栏显示提示,或使用闪烁的图标表示错误——这样用户在方便的时候进行处理。
- 当错误很明显时,直接改正它。例如当用户保存文件时,如果当前驱动器已满,则检查其他驱动器的空间,如果有可用空间则保存文件;同时在状态栏上显示信息,告诉用户程序都做了什么。
- 并非所有的错误都需要立即解决,这时可以采用将错误记录在日志文件中的方法,供用户方便时浏览。例如在 Windows 2000 中,如果登录时网络连接出现问题,则将问题记录在系统日志中,用户可以通过系统信息管理器查询。

1.1.2 Windows 界面规则

Windows 操作系统的一个重要优势在于它为所有应用程序提供了相同的界面。这样,一个有经验的用户能够很快掌握原先并没有使用过的 Windows 应用程序。菜单就是一个很好的例子:大多数 Windows 应用程序遵循“文件”菜单在最左边,其后是“编辑”、“工具”菜单,最右边是“帮助”菜单的规则。可能有人认为“文档”要比“文件”更合适,或“帮助”菜单应该排在最左边,并且完全可以按照这些想法设计应用程序的界面。然而,这样就会给用户带来不必要的混乱,而且当用户在切换到另一个应用程序时,还必须调整一下使用方式,这样就降低了程序的可用性。此外菜单命令的位置也很重要。例如,用户希望在“编辑”菜单下找到“拷贝”、“剪切”和“粘贴”命令。总之,除非有充分的理由,否则不应该改变公认的 Windows 界面规则。使用已经存在的界面设计规则,不但能够使应用程序的可用性更好,而且会使自己的设计有个很好的开端(也就是说,用户界面的布局有个大概的框架)。

检测用户界面可用性最好的手段,就是使用户自始至终地参与到设计过程中。可用性检测的一个关键就是用户是否能够较为容易地了解如何使用应用程序的功能。例如在 Windows 3.1 中,很少有用户知道使用 Alt + Tab 键能够在应用程序间进行切换,因为几乎没有什么线索能够帮助用户发现这一点。要测试这一点,可以要求用户完成某个任务而不加任何解释(例如,要求用户从信函模板创建一个新文档)。如果他们不能完成,或中间经历了多次尝试,那么就需要重新考虑设计了。

1.1.3 界面布局原则

应用程序的布局不仅影响其外观,而且对其本身的易用性有着举足轻重的作用。这包括控件的位置、元素之间的协调性、空间的使用以及设计的简单性等。

1. 控件位置

在大多数界面设计中,并非所有的组成元素都具有同等的重要性,因此必须保证常用的重要元素处于最明显的位置。

绝大多数语言都具有从左到右和从上到下的书写顺序。因此当用户观看屏幕时,会从左上角开始——最重要的元素应该放置在那里。例如,如果在屏幕上显示包括消费者信息的表单,则应该首先显示姓名字段;而诸如“确定”或“下一步”之类的按钮,则应该在屏幕右下方显示。这是因为,通常用户只是在完成了整个表单后,才会单击这些按钮。

将元素和控件分组也非常重要。将信息、功能相近或关联的控件分组排列,要比分散排列好。分组排列大多是通过组框等控件完成的。

2. 协调性

用户界面必须保证其协调性。协调的外观和感觉将使应用程序看起来很舒服。相反,缺乏协调性,会使应用程序看起来很混乱,从而使用户低估其功能和稳定性。

为了可视协调性,需要在开发前有一些大概的规划。例如,控件的种类、尺寸以及字体等等。这时先做一个模型将会有助于后续的开发。

由于 Visual C++ 提供了多种类型的控件,这很容易导致你产生将各种类型的控件都加以使用的想法。在开发中必须注意避免出现此类想法,而应该仔细选择最适合应用程序的一些控件。例如列表框、组合框、列表视图控件和树视图控件都能用于显示一系列信息,但是对于开发者来说,应该尽可能地选择单一风格的控件。

此外,还需要注意所用的控件是否合适。例如将编辑框设置为只读用来显示文本不如使用静态文本控件更合适。保证一致性对于界面设计也十分重要,例如如果在某处将允许用户编辑的文本属性设置为黑底白色,除非有充分的理由,不要在其他类似之处使用其他颜色配置。

出于易用性的考虑,在应用程序的不同窗口中也需要保证一定的一致性。如果在应用程序的一个对话框中使用灰色背景,而在另一个对话框中使用白色背景,这会使整个应用程序看起来缺少内部的联系。总的原则就是选择什么风格就坚持什么风格,即使在某些情况下需要重新设计某些功能。

3. 一致性

抽象地说一致性就是某个对象的可视化线索,实际的例子到处都是。例如,自行车上的把手就是手扶的地方。又比如,在 Windows 应用程序中,“打开”工具栏按钮图标就是用于打开文件夹的。通俗地讲,一致性就是看到其外观就能猜到其功能。

用户界面也需要使用一致性。例如,应用程序中的具有三维效果的按钮表示它们可以被按下;而同时又存在具有平坦风格的按钮,那么就失去了一致性。用户无法确定一个按钮到底是否是命令按钮。有些情况下平坦风格的按钮可能会更合适,例如游戏或多媒体应用程序。但是,在应用程序中保持一致性是压倒一切的。

编辑框控件也提供某种方式的一致性。一般来说,允许用户进行编辑的控件具有边界和白色背景。当然你完全可以将控件的风格设置成为无边界(这就有些像静态控件),这时用户就不会很快意识到它是可以被编辑的。

4. 空间的使用

应用程序中空间的使用也很重要,它有助于改善程序的外观和对某些元素的强调。当然空间并非一定要使用。然而过多的控件拥挤在一起,会增加寻找的时间降低效率。在设计时,需要综合考虑以确定最佳的分布。

使控件间距一致,以及控件垂直和水平对齐也会提高应用程序的易用性。这与报纸中文字的有序排列会方便阅读的道理一样。在 Visual C++ 中提供了一些工具用于调整布局,当读者使用资源管理器时会经常用到。

5. 简单性

界面设计中最重要准则可能就是简单性了。简单即美,这也是艺术中的一个准则。如果应用程序的界面过于复杂,也会使用户望而却步。

用户界面设计中经常出现的一个误区是试图以真实世界中的对象作为设计的模型。例如要设计一个用于完成保险表单的应用程序,很自然的反应就是创建一个实际表单的翻版作为界面。而这将会导致一些问题:纸的尺寸和形状与屏幕上的不同,完全照搬就会使界面元素限于文本框和复选框,而这并没有为用户带来任何方便。

最好的方法是设计自己的界面,例如设计打印功能界面。通过逻辑关系创建不同的标签或链接表单,从而使用户无需滚动翻页即可得到全部信息。此外还可以使用列表控件为用户提供选择,这样能够减少用户的输入量。只要多从用户的角度考虑,就会设计出相对简单易用的界面。

简化界面的另外一个有效方法,就是根据某个功能的使用频率决定其显示方式。例如 Word 2000 中会智能地根据某个按钮(控件)的使用频率决定其显示与否(位置)。有时提供默认值也会简化界面。例如,如果大多数用户都喜欢使用加粗的文本,那么将加粗风格设置为默认值,会比用户自己主动选择要方便一些。

使用向导也能简化复杂的操作,这一点使用 Windows 软件较多的读者都深有体会。

测试应用程序简单性的最好方法就是实际使用该程序。如果在没有帮助的情况下,普通用户不能很快完成某个操作,那么就需要考虑重新设计了。

6. 颜色

多彩的界面会增加应用程序的吸引力。现在使用的绝大多数显示器能够显示数百万种颜色,这往往会导致用户对颜色的过度使用。颜色的效果是千差万别的,用户的喜好也


会因人而异。因此在设计应用程序时,需要注意不同文化中颜色的差异。一般来说,在颜色的使用上应该持保守的态度,尽量选择软色调和中性颜色。

当然,对颜色的选择也需要考虑用户的情况。例如,在设计儿童软件时,亮红色、绿色和黄色是很好的选择;而对于金融或银行应用程序,这些颜色会产生反面影响。使用少量的亮色能够有效地突出重要区域。作为原则,在应用程序中应该限制颜色的数量,而且应该保持前后的一致性;应该尽可能使用标准 16 色调色板,这样会扩大应用程序的兼容性;此外,色盲用户也是需要考虑进内的。

7. 图片和图标

在应用程序中使用图片和图标,也能提高应用程序的吸引力。图片所传达的信息有时是文字所远远不及的,同样,其致命的弱点在于不同人的理解可能也会不同。

工具栏按钮上的不同图标,形象地表达了不同的功能。但是当用户不能确定某个图标的功能时,就会产生反作用。在设计工具栏图标时,应该参考其他应用程序以得到设计的标准。例如绝大多数 Windows 应用程序,会将打开文件夹的图像作为“打开”命令工具栏按钮图标。如果你的应用程序违反了标准,就会给用户带来混乱。

此外还需要考虑文化的差异,例如美国应用程序经常将  作为“邮件”按钮的图标。而在其他国家中用户却并不将该图标看作邮筒,从而对按钮的功能理解出现偏差。

在设计应用程序时,应尽量使用 16 色调色板图像,以提高应用程序的兼容性。

8. 字体

字体是用户界面的重要组成部分,因为它们通常用于与用户进行重要信息的交互。在选择字体时,必须保证它能在各种不同分辨率的显示器上都有良好的可读性。

除非计划将应用程序与字体一同发行,否则最好使用标准 Windows 字体,例如 Arial、New Times Roman 或 System 字体。如果用户的系统不包括指定的字体,则系统将使用其他字体,这就有可能造成实际的显示与预期不一致。例如,CorelDraw 9.0 的英文版在 Windows 9x 下字体就无法完全显示,造成的后果就是文字不全。如果应用程序将在世界范围内发布,则需要为每种语言选择合适的字体。此外还需要考虑字体所占的空间,例如中文比英文所占的空间大 50%。这里还要旧话重提,一定要保持一致性,类似功能的文本要使用相同的字体。

1.1.4 用户辅助模型

无论使用多么优秀的用户界面,用户总会需要帮助。应用程序的用户辅助模型包括在线帮助、帮助手册、工具提示、状态栏、“这是什么?”以及向导等。

与应用程序其他部分的设计一样,用户辅助模型的设计也应该在开发前进行。模型的内容则与应用程序的复杂程度以及预期的用户有关。

在线帮助是应用程序的重要组成部分——它通常是用户寻找问题答案的首选。在设计帮助系统时,要时刻牢记它的基本用途是回答问题。当创建主题和索引时,一定要从用

户的角度出发。例如,“应该如何格式化页面?”要比“编辑、页面格式菜单”更容易定位问题。此外,一定要保证上下文敏感性,也就是说当用户选择了“格式”然后按下 F1 键,那么应该出现的是针对这个词的帮助,而不是出现帮助主题。印刷或电子的帮助手册是另一种非常有用的工具,它们能提供在简洁帮助主题中难于传递的信息。此外,工具提示、状态栏等也能够为用户提供很大的方便(使用 Visual C++ 能够很轻易地为应用程序添加此类支持)。

1.2 Windows 编程机制

要更好地使用 VC++ 进行 Windows 编程就需要了解其内部运行机制,这主要包括消息驱动和 MFC 类库。

1.2.1 消息驱动

进行 Windows 编程之前,必须了解一些 Windows 的运行机制。在 Windows 操作系统中,应用程序主要是以窗口的形式存在。消息传递是 Windows 操作系统和应用程序间、不同的应用程序间互相交流的主要形式。每一个应用程序都对相应的消息作出执行动作,这种执行动作称为消息响应。

消息的产生和发送以队列的形式进行。消息响应遵循一定的顺序。MFC 类库为这种消息响应机制提供了比较完整的处理功能。构成 MFC 类库的很多基类都具有处理相应消息的函数。在 DOS 等控制台模式程序中,对诸如鼠标、键盘等的控制是通过轮询(分别定时查询这些设备的输入请求)完成的。而在 Windows 环境中,这些控制是通过消息完成的,因此 Windows 也被称为“基于事件的,消息驱动的”操作系统。在编写 Windows 程序时,必须保证与 Windows 平台下同时运行的程序协同合作。

在 Windows 应用程序中,窗口既是一个可视界面,也是应用程序的控制消息发送接收端。窗口不仅提供了可视化的应用程序的命令,也是 Windows 消息的产生和响应的地方。图 1-2 所示为一个标准的 Windows 窗口,其中包括客户区和非客户区。非客户区包括窗口的边框、菜单和标题区。客户区是窗口中除了非客户区剩下的部分。通常情况下,窗口的非客户区由 Windows 维护,客户区由应用程序进行管理和操作。Windows 下的窗口有许多类型,如对话框、属性页、控件台模式的 DOS 命令框、工具栏和菜单栏等。甚至按钮、列表框等控件也可以被看作一个窗口,实际上在 MFC 类库中,这些控件的管理类几乎都是由 CWnd 类(窗口管理类)派生出来的。每一种类型的窗口都有一套适于它们自己的公用操作,如移动、改变大小、隐藏、显示、允许使用和禁止使用等。

在 Windows 中,每个应用程序都是基于事件和消息的,而且包含一个主事件循环。该循环持续反复检测是否有用户事件发生。每次检测到一个用户事件,程序就对之作出响应。这些事件包括移动鼠标指针、按键、单击或双击按钮等。当 Windows 接收到这些事件后,会产生一些相应的消息。应用程序接到这些消息后,按每一个消息的专门用途,产生

一系列的执行动作。这些执行动作称之为响应。响应包括重画窗口、改变窗口大小、关闭窗口、打开一个位图、调用一个动态库等。Windows 可以处理 3 种不同类型的消息：

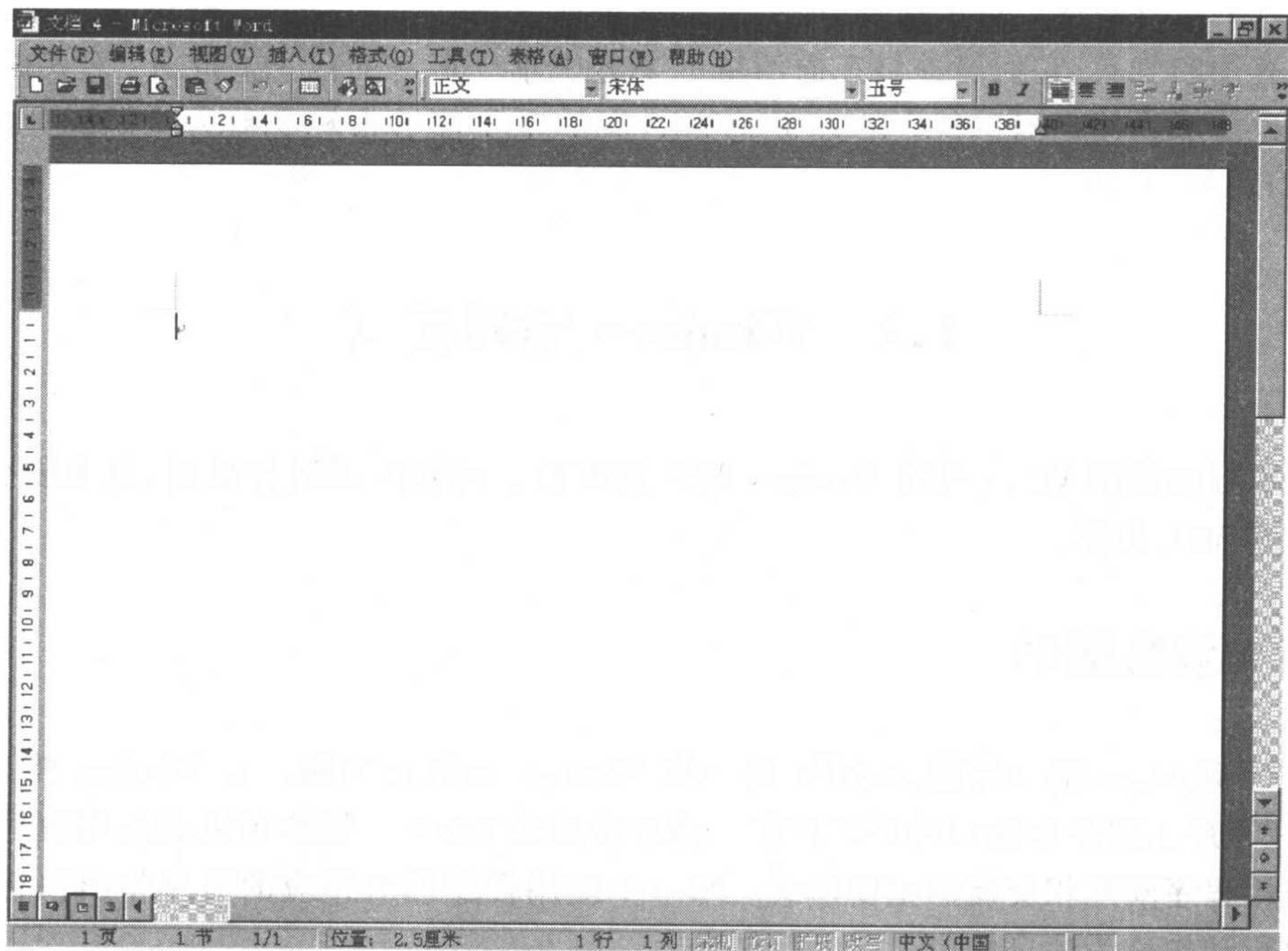


图 1-2 标准的 Windows 窗口

- 标准 Windows 消息

标准 Windows 消息由窗口和视图来处理,这类消息通常含有用于确定如何对消息进行处理的一些参数,这些参数通常使用 WM 前缀,例如 WM_PAINT 等。这些消息可以用来告诉程序它应该启动或关闭,也可以用来告诉窗口它正被改变大小或移动。

- 控件通告消息

控件通告消息包括按下按钮或输入字符等事件消息。同标准 Windows 消息一样,控件通告消息由窗口和视图处理。例如,当用户对编辑控件中的文本作出了修改后,编辑控件向其父窗口发送的 WM_COMMAND 消息中包含了 EN_CHANGE 控件通告码。窗口的消息处理函数将对该通告消息作出合适的处理,例如接收输入到控件中的文本。

控件通告消息与其他标准 Windows 消息一样在框架消息循环中被处理。但是有一个例外,这就是当用户单击按钮时,按钮所发送的 BN_CLICKED 控件通告消息被当作命令消息处理,并与其他命令消息一起进行消息循环。

- 命令消息

命令消息包括来自用户界面对象,如工具栏按钮、菜单命令或快捷键等的 WM_COMMAND 通告消息。命令消息与其他消息的处理不同,它可被更广泛的对象(如文档、文档模块、应用程序模块、窗口和视图等)处理。如果某条命令直接影响到某个对象,则由该对象来处理这条命令。

如果有必要的话,用户也可以自定义消息用于特定操作使用。用户可以通过消息的前缀来判断产生消息的对象类型,如表 1-1 所示。

表 1-1 Windows 消息前缀和对象类型

消息前缀	对象类型
ABM、ABN	系统菜单消息
ACM、CAN	动画控件消息
BM、BN	按钮控件消息
CB、CBN	组合框控件消息
CDM、CDN	普通对话框消息
CPL	控制面板应用程序消息
DBT	设备更改消息
DL	列表框拖动消息
DM	对话框消息
EM、EN	编辑框消息
FM、FMEVENT	文件管理器消息
HKM	标题控件消息
IMC、IMN	热键控件消息
LB、LBN	IME 窗口消息
LVM、LVN	列表框控件消息
NM	列表视图消息
PBM	父窗口通告消息
PBT	进度条控件消息
PSM、PSN	属性表消息
SB	状态栏消息
SBM	滚动条消息
STM、STN	静态控件消息
TB、TBN	工具栏消息
TBM	轨迹条消息
TCM、TCN	标签控件消息
TTM、TTN	工具提示消息
TVM、TVN	树形控件消息
UDM	上下控件消息
WM	一般窗口消息

注意 表中有些消息前缀是成对出现的,例如 TCM 和 TCN。如果以 M 结尾表示消息是发送给控件的,而如果以 N 结尾则表示消息是传递给窗口的。

所有在 Windows 环境下的编程,都必须谨慎而恰当地处理消息。在 Windows 中,消息是独立的,是由大量的物理数据组成的,很容易被排列和优先化。消息独立于特定的语言或处理器类型,这使得基于消息的程序能很容易地移植到不同机器的 Windows 环境下。对 Windows 消息的相同处理机制,使包括 Visual C++ 在内的一系列可视化编程得到长足的发展。

消息在 Windows 环境下是按一定的顺序发送的。程序的细微差别能够导致消息以不同的顺序发送和接收。Visual C++ 程序员要对消息的排序和处理施加一定的控制。当一个应用程序生成时,消息在外部环境产生后,传送到应用程序框架的内部,消息处理机制将在应用程序的框架内部完成。但这种消息处理机制并不是对所有的消息都加以响应。当消息队列中的消息依次传送到应用程序框架内部时,应用程序将先对这些消息进行判断,遇到相应的消息时才进行处理,否则将跳过这一条消息,继续运行。在没有 MFC 类库之前,为了达到这种继续运行的功能,必须编制大量的程序代码。MFC 类库出现后,这种功能被模块化固定在类的功能中。

1.2.2 MFC 类库

MFC(Microsoft Foundation Class)类库是用来编写 Windows 应用程序的 C++ 类集,该类集以层次结构组织,其中封装了大部分 Windows API 函数,所包含的功能涉及到整个 Windows 操作系统。MFC 不仅为读者提供了 Windows 图形环境下应用程序的框架,而且提供了创建应用程序的组件。使用 MFC 类库和 Visual C++ 提供的高度可视的应用程序开发工具,可使应用程序的开发变得更简单,开发周期极大地缩短,而在代码的可靠性和可重用性上得到很大的提高。

在 Visual C++ 2.0 以后的版本中,Microsoft 公司已经推出了 MFC 类库,并在 MFC 类库中提供了一种用于基本 Windows 应用程序的结构。在面向对象的程序设计中,MFC 可以提供一个功能完整的应用程序框架,使程序员可以将其方便地扩展为一个完整的 Windows 应用程序。这使得程序员不必从头重新设计,从而节省了大量的开发时间。而且 MFC 类库所提供的大量的以类为基础的代码块可以指导编程时某些技术和功能的实现。

实际上,程序员和 Microsoft 公司都可从 MFC 类库的这些优点中获益。程序员借助于 MFC 类库已经搭建好的应用程序框架,就可按照标准的 Windows 界面编制程序,从而有效地提高了效率。对 Microsoft 公司来说,新的应用程序能立即支持所有的标准 Windows 特征,其中包括很多高级特征,用普通、明确的定义方式简化了过去繁复的重复开发。

MFC 类库的优越性表现在以下几个方面:

- MFC 几乎完整地封装了 Windows API 函数。MFC 库为经常使用的 Windows API 函数提供支持,包括窗口函数、消息、控件、菜单、对话框、GDI 对象、对象链接,以及多文档界面等。此外 MFC 类库也提供了具有共性的应用程序的操作,如打印、打印预览、状态条、工具条、数据支持和 OLE 支持等。
- MFC 支持多线程,所有的应用至少有一个线程,这个线程由 CWinApp 类的对象创建,被称为“主”线程。为方便多线程编程,MFC 还提供了同步对象类。
- MFC 库具有同以 C 语言为基础的使用 Windows API 开发的 Windows 应用程序的共存能力。在同一程序中,程序员可以同时使用 MFC 的类和 Windows API 调用。
- MFC 库提供了自动消息处理功能。MFC 库消除了最易产生错误的来源,即 Windows API 的消息循环。MFC 库将自动处理每一个 Windows 消息,且每一条 Windows 消息被直接映射到一个进行处理的成员函数。

- MFC 类库使用与 Windows API 相同的命名约定,这使得程序员根据类名就可以知道类的功能。

应用程序从 MFC 类库中派生时必须有一些基本元素。下面是 MFC 类库中最重要的类的说明。

1. CObject 类

CObject 类是 MFC 的抽象基类,MFC 库中的大多数类是从 CObject 类派生出来的。它是 MFC 中多数类和用户自定义子类的根类。该类为程序员提供了希望溶入所编程序的许多公共操作。这些操作包括:对象的创建和删除、串行化支持、对象诊断输出、运行时信息以及集合类的兼容等。CObject 类的声明在头文件 `afx.h` 中。

2. 应用程序结构类

该类用于构造框架应用程序的结构,它提供多数应用程序公用的功能。编写程序的任务是填充框架,添加应用程序专用的功能。

(1) 应用程序和线程支持类

CWinThread 类是所有线程类的基类,窗口应用程序类 CWinApp 类就是从该类中派生而出的。每个应用程序有且只有一个应用程序对象,在运行程序中该对象与其他对象相互协调,该对象从 CWinApp 类中派生出来。该类封装了初始化、运行和终止应用程序的代码。

(2) 命令相关类

CCmdTarget 类是 CObject 的子类,它是 MFC 库中所有具有消息映射属性的类的基类。消息映射规定了当一对象接收到消息命令时,应调用哪一个函数对该消息进行处理。程序员很少需要从 CCmdTarget 类中直接派生出新类,往往都是从它的子类中派生出新类,如窗口类(CWnd)、应用程序类(CWinApp)、文档模板类(CDocTemplate)、文档类(CDocument)、视类(CView)及框架窗口类(CFrameWnd)等。

(3) 文档类

文档对象由文档模板对象创建,管理应用程序的数据。视对象表示一个窗口的客户区,显示文档数据并允许读者与之交互。有关文档/视结构的类如下:

- CDocTemplate 类:文档模板的基类。文档模板是协调文档、视和框架窗口的创建。
- CSingleDocTemplate 类:单文档界面(SDI)的文档模板。
- CMultiDocTemplate 类:多文档界面(MDI)的文档模板。
- CDocument 类:应用程序专用文档的基类。

3. 可视对象类

(1) CWnd 类

该类提供了 MFC 中所有窗口类的基本功能。它是 CCmdTarget 类的子类,创建 Windows 窗口要分两步进行:首先,执行构造函数,构造一个 CWnd 对象,然后调用 Create 创建 Windows 窗口并将它连到 CWnd 对象上。MFC 中还从 CWnd 类派生出了进一步的窗口类

以完成更具体的窗口创建工作,这些派生类有:

- CFrameWnd 类: SDI 应用程序主框架窗口的基类。
- CMIDFrameWnd 类: MDI 应用程序主框架窗口的基类。
- CMDIChildWnd 类: MDI 应用程序文档框架窗口的基类。

(2) CView 类

使用 CView 类可以在其他窗口中创建子窗口,它可以提供一个特殊的接收外来输入的结构窗口。也就是说视图类控制用户如何观看文档的数据,以及怎样与这些数据交互。即 CView 类管理着框架窗口的客户区,为用户与 Windows 之间提供可视接口,该类接收来自用户的键盘或鼠标输入,还允许用户对数据预览和打印。

CView 类广泛用于基于文档的应用程序中,利用该类的派生类可图形化地管理文档数据,并将用户对文档的操作通过它来实现。CView 类可用来实现用户自定义视图类的基本功能。在程序运行时视图类用于视图的实现。一个视图只能分配给一个文档,但是一个文档可以拥有多个视图。

如果用户需要滚动显示,可由 CScrollView 实现。如果要视图具有来自对话框模板资源的用户界面,可由 CFormView 实现。对普通文本数据,使用 CEditView 或由 CEditView 实现。对有格式的数据存取应用程序,如数据输入程序,由 CRecordView (ODBC 中)或 CDaoRecordView (DAO 中)实现。其他可利用的视图类还有 CTreeView、CctrlView、COleDBRecordView、ChtmlView、CListView 和 CRichEditView 等。

(3) CDialog 类

由于对话框是一个特殊的窗口,所以该类是从 CWnd 类中派生出来的。对话框子层次结构包括通用对话框类 CDialog 以及支持文件选择、颜色选择、字体选择、打印、替换文本的公共对话框子类。这些子类包括:

- CFileDialog: 提供打开或保存一个文件的标准对话框。
- CColorDialog: 提供选择一种颜色的标准对话框。
- CFontDialog: 提供选择一种字体的标准对话框。
- CPrintDialog: 提供打印一个文件的标准对话框。
- CFindReplaceDialog: 提供一次查找并替换操作的标准对话框。

CDialog 类可用于创建模式对话框和无模式对话框模型,是该子层次结构的根。

(4) 菜单类 CMenu

该类是 CObject 类的子类,用于管理菜单。它是 Windows HMenu 的封装,提供了与窗口有关的菜单资源创建、修改、跟踪及删除成员函数。

(5) 控件类

控件子层次结构包括若干类,使用这些类可创建静态文本、命令按钮、位图按钮、列表框、组合框、滚动条、编辑框等。这些直观控制为 Windows 应用程序提供了各种输入和显示界面。

- CStatic 类: 静态文本控件类,常用于管理标注、分隔对话框或窗口中的其他控件。
- CButton 类: 按钮控件类,该类为对话框或窗口中的按钮、检查框或单选按钮提供一个总的接口。

- CEdit 类：编辑控件类，该控件类用于管理用户的文字输入。
- CRichEditCtrl 类：富文本编辑控件类，除了编辑控制的功能外，还支持字符和图形格式，以及 OLE 对象。
- CScrollBar 类：滚动条控件类，该类提供控制条的功能，用作对话框或窗口中的一个控制，用户可通过它在某一范围内定位。
- CProgressCtrl 类：进展指示控件类，用于管理操作的进度。
- CSliderCtrl 类：滑块控件类，该类用于管理滑标移动选择。
- CListBox 类：列表框控件类，该类用于管理列表框的显示和选择。
- CComboBox 类：组合框控件类，该类用于管理组合框的操作。
- CBitmapButton 类：位图按钮类，该类用于管理带有位图而非文字标题的按钮。
- CSpinButtonCtrl 类：上下控件类，该类用于管理上下控件，该控件带有一个双向箭头按钮，单击某个箭头按钮增大值或减小值。
- CAnimateCtrl 类：动画显示控件类，用于管理动画播放。
- CToolTipCtrl 类：工具提示管理类，工具提示是一个小的弹出式窗口，显示一行文本，描述应用程序中一个工具的作用。
- CHotKeyCtrl 类：热键控件类，该类用于管理热键快速执行某项操作。

(6) 控制栏类

CControlBar 类为工具栏、状态栏、对话框条和分割窗口创建模型。该类是 CToolBar、CStatusBar、CDialogBar 的基类，负责管理工具条、状态条、对话框的一些成员函数。控制条指的是连接在主窗口框架的顶部或底部的小窗口。

- CStatusBar 类：状态条控制窗口的基类。
- CToolBar 类：包含非基于 HWND 的位图式命令按钮的工具条控制窗口。
- CDialogBar 类：控制类似于工具栏的非模式对话框。

4. 绘图和打印类

用于管理绘图和打印的类包括以下几种：

(1) CGdiObject 类

图形绘画对象类以 CGdiObject 类为根类，可用于创建绘画对象模型，如画笔、刷子、字体、位图、调色板等。这些子类包括：CPen、CBrush、CFont、CBitmap、CPalette、CRgn 等。

- CBitmap 类：该类封装了 GDI 位图，提供操作位图的接口。
- CBrush 类：该类封装了 GDI 画刷，可被选择为设备描述表的当前画刷。
- CFont 类：该类封装了 GDI 字体，可被选择为设备描述表的当前字体。
- CPalette 类：该类封装了 GDI 调色板，用作应用程序和彩色输出设备如显示器之间的接口。
- CPen 类：该类封装了 GDI 画笔，可被选择为设备描述表的当前画笔。
- CRgn 类：该类封装了 GDI 域，用于操作窗口内的椭圆域或多边形域。该类一般与 CDC 类的裁剪成员函数一起使用。

(2) CDC 类

设备环境类及其子类支持设备描述表对象,是 CObject 类的子类。CDC 类是一个较大的类,包括许多成员函数,如映射函数、绘画工具函数、区域函数等,通过 CDC 对象的成员函数可以完成所有的绘画工作,它的子类有: CClientDC、CWindowDC、CPaintDC 和 CMetaFileDC。

- CPaintDC 类: 显示设备环境类,用于窗口的 OnPaint 成员函数和视的 OnDraw 成员函数中,自动调用 BeginPaint 进行构造,调用 EndPaint 进行析构。
- CClientDC 类: 窗口客户区的显示设备环境类。例如,用于在快速响应鼠标事件时进行绘画。
- CWindowDC 类: 整个窗口的显示设备环境类,包括客户区和框架区。
- CMetaFileDC 类: Windows 元文件的设备环境类。Windows 元文件包含一个图形设备接口(GDI)命令序列,该序列可被重新执行而创建一幅图像。对 CMetaFileDC 的成员函数的调用记录在一个元文件中。

5. 通用类

通用类提供了许多通用服务,例如文件 I/O、诊断和异常处理,此外还提供处理通用数组和列表等功能。

(1) 文件类和输入/输出类

文件类和输入/输出类用于文件处理。利用这些类可以操作各种文件,它们对文件所能实施的操作远远超过文档自动化对于文件的操作。其中的文件输入/输出类(File I/O Classes)为传统的磁盘文件、内存文件、ActiveX 流和 Windows Sockets 读写提供界面。所有的 CFile 类派生类都可被 CArchive 类使用进行串行化。如果想编写自己的输入/输出处理,可以使用 CFile 类和 CArchive 类,一般不必再从这些类中派生新类。如果使用应用程序框架,则只需提供关于文档如何将其内容串行化的详细信息,File 菜单上的 Open 和 Save 命令的默认实现将会处理文件输入/输出(使用类 CArchive)。

- CFile 类: 该类提供访问二进制磁盘文件的总接口。
- CMemFile 类: 该类提供访问驻内存文件的总接口。
- CStdioFile 类: 该类提供访问缓存磁盘文件的总接口,通常采用文本方式。
- CArchive 类: 该类与 CFile 对象一起通过串行化实现对象的永久存储。

(2) 异常类

- CException 类: 该类是所有异常处理类的基类。程序员不能直接创建 CException 对象,只能创建派生类的对象来捕获指定的异常情况。
- CArchiveException 类: 该类管理归档异常。
- CFileException 类: 该类管理文件处理异常。
- CMemoryException 类: 该类管理内存异常。
- CNotSupportedException 类: 该类管理使用未支持特征产生的异常。
- CResourceException 类: 该类管理装载 Windows 资源失败产生的异常。
- CUseException 类: 该类管理停止用户操作产生的异常。

(3) 数据类型类

数据类型类可以将多种对象存放到数组、列表和“映射”中。但这些数据类是模板,它们的参数确定了存放在集合中的对象类型。CArray、CMap 和 CList 类使用全局帮助函数,帮助函数通常需要定制。类型指针类是类库中其他类的包装类,利用这些包装类,应用程序可借助于编译器的类型检查以避免出错。

- CArray 类: 该类将元素存储在数组中。
- CMap 类: 该类将键映射到值。
- CList 类: 该类将元素存储在一链表中。
- CTypedPtrList 类: 该类将对象指针存储在一链表中。
- CTypedPtrArray 类: 该类将对象指针存储在一数组中。
- CTypedPtrMap 类: 该类将键映射到值,键和值都为指针。

6. 数据库类

数据库类支持访问数据库中的数据源,它包括 ODBC(Open Database Connectivity)数据库和 DAO(Data Access Object)数据库支持。这两类的根类都继承自 MFC 类库的总根类 CObject。通过 ODBC 数据库中的类可开发数据库应用来访问多种数据库文件。该层次结构中主要包括的类有:

- CDataBase 类: 封装对数据源的连接,通过此连接应用程序可对该数据源进行操作。
- CRecordset 类: 封装从数据源选出的记录。
- CRecordView 类: 提供直接连接记录集对象的格式视图。
- CFieldExchange 类: 提供上下文信息,支持记录字段交换,即在字段数据成员、记录集对象的参数数据成员及数据源上的对应列表之间交换数据。
- CLongBinary 类: 封装存储句柄,用于存储二进制大对象,例如位图。
- CDBException 类: 封装数据存取处理过程中的失败产生的异常。

7. 网络工作类

网络工作类提供了编写 Internet 服务器程序所需的基本类,封装了 Win32 Internet (WinInet)和 ActiveX 技术,从这些类派生出的新类就可以根据需要开发 Internet 服务器程序。它包括以下几个组成部分:

- ISAPI 类。
- Windows Socket 类。
- Win32 Internet 类。

8. OLE 类

对象连接与嵌入(OLE)子层次结构提供了 9 个类,这 9 个类分为三大类:普通类、客户类和服务器类。其中 COleDocuemnt、COleItem、COleException 为支持 OLE 的普通类,COleClientDoc、COleClientItem 为支持 OLE 的客户类,COleServer、COleTemplate、COleServerDoc、COleServerItem 为支持 OLE 的服务器类。这些类支持 OLE 的所有功能。

1.2.3 MFC 框架与消息处理

用 MFC 编制的应用程序遵循标准结构,保证了整个应用程序的基本运行模式。在 Visual C++ 中,AppWizard 工具可以根据用户规定的参数,选择创建框架应用程序需要的类。这种框架结构提供了用户应用程序的框架,定义了实现 Windows 用户应用程序所需界面的基本设置。

如果从 MFC 类库中利用 AppWizard 工具派生出应用程序,就把大量的消息处理分配给应用程序框架提供的代码来处理,使应用程序框架完成了大部分用户界面的管理工作。应用程序框架截取 Windows 向应用程序发出的消息,并根据需要加以处理。利用重载函数,用户可以响应所需要的消息,但很多不须直接处理的消息将由框架处理。在这些过程中,消息是在外部环境中产生的,而处理控制是在框架内部完成的,这就是经典的事件驱动编程。

应用程序框架提供的内容非常丰富,它从 MFC 类库中的类 CWinApp 继承了大部分性能,获得了 Windows 应用程序所需的大部分基本函数,可以初始化、运行和终止应用程序。

在 Windows 操作系统下,应用程序所做的大部分是对消息进行响应,而这些响应几乎都是基于处理 Windows 消息的。MFC 类库为窗口的消息处理提供了框架。这些从 CCmdTarget 类派生出来的类能够拥有自己的消息映射,这种功能使 MFC 类库的消息处理更为简单,并能够加强类的功能性封装,避免了使用类时的重复性操作。

MFC 类库为了进一步扩展重复使用性,为大多数 Windows 应用程序中用到的范围很广的命令提供了默认操作,这些默认操作中大部分同时也被包含在由 AppWizard 产生的默认菜单中,表 1-2 列出了 MFC 类库中有默认操作的菜单命令。这些由 AppWizard 创建的菜单命令消息在 AFXRES.H 中定义,它们的命名规则一般是“ID_ + 菜单名 + 命令名”。如果想执行这些菜单命令,可以从应用程序中的任何一处发送一条预定义的命令消息,MFC 类库就会进行默认处理。程序员也可以对这些命令进行重载、编辑,以实现自己定义的功能。

表 1-2 MFC 类库中默认操作的菜单命令

菜单项	默认菜单命令
文件菜单	New, Open, Close, Save, Save, Page Setup, Print Setup, Print, Print Preview, Exit
编辑菜单	Clear, Clear All, Copy, Cut, Find, Paste, Repeat, Replace, Select All, Undo, Redo
视图菜单	Toolbar, Status Bar
窗口菜单	New, Arrange, Cascade, Tile Horizontal , Tile Vertical, Splite
帮助菜单	Index, Using Help, About

应用程序的构造过程被执行之后,CWinThread 类的 Run 函数为应用程序创建了一个线程,并提供了一个消息循环。消息循环的唯一功能是等待消息,并将其发送到应该接受处理的消息入口。消息的发送和接收具有一定的查询顺序。当消息循环接收到一条 Windows 消息时,必须通过查询一种内部结构来确定消息要发送的窗口。这种内部结构

把当前所有的窗口映射到对应的窗口类。MFC 基类还检测这些目标类是否为这条消息提供了消息入口。如果找到入口,消息送到处理的程序中,否则 MFC 类库将进行消息映射检测,沿着层次向上移动,直到找到入口为止。

这种查询检测机制对命令消息来说较为复杂。在大多数情况下,命令目标将以下面的顺序发送命令:当前活动的子命令目标对象、命令目标本身和其他命令目标。表 1-3 列出了常用命令的详细消息传送路径。

表 1-3 消息传送路径

对象类型	消息发送顺序
MDI 主框架	1. 活动的 CMDIChildWnd 2. 本框架窗口 3. 应用程序(CWinApp)
MDI 子框架	1. 活动的视图 2. 本框架窗口 3. 应用程序
SDI 主框架	1. 活动的视图 2. 本框架窗口 3. 应用程序
视图	1. 本视图 2. 与本视图相连的文档
文档	1. 本文档 2. 文档
对话框	1. 本对话框 2. 拥有本对话框的窗口 3. 应用程序

1.3 Windows 应用程序结构体系

绝大多数 Windows 应用程序都是以文档/视图(Document/View)结构为基础的,文档用以管理应用程序的数据,而视图用以显示文档并处理其与用户的交互。本节将向读者介绍文档/视图结构编程的基本知识。

1.3.1 文档/视图结构概述

文档类和视图类在一个典型的 MFC 类库构成的应用程序中是成对出现的。数据存储在文档里,但是视图类有权利访问到这些数据。文档类和视图类的划分把数据的存储和维护与它的显示分开来。从视图类访问文档类中的数据需要一些方法。这些方法包括:

- 调用 `GetDocument` 函数, 获得一个指向文档的指针。
- 使视类成为文档类的一个友元类。

使用上面的方法后, 当视图类准备重绘窗口或使用数据时, 就能够通过这些途径操作数据。例如, 在视图类的 `OnDraw` 函数里, 视图类可以通过调用 `GetDocument` 函数获得一个文档类的指针。然后它可以使用那个指针访问文档类里的一个 `CString` 类型的数据。视图类把字符串传递给 `TextOut` 函数, 用户就可以看到在窗口输出了一串字符串。视图类也可以通过响应鼠标来选择或编辑数据。这里假设用户是在一个可以管理文本的视里键入了一串字符串。视图类获得一个指向文档的指针, 并使用指针把新数据传递给文档, 文档将以某种结构存储数据。在一个文档对应多个视的应用程序中(如在一个文本编辑器中切分窗口), 视首先将把新数据传递给文档, 然后视将调用文档的 `UpdateAllViews` 成员函数, 通知此文档中所有其他的视更新数据。这个功能可以使所有的视同步。

使用 MFC 类库中的文档/视图类结构最大的好处在于: 这种结构在一个文档中支持多个视。假设应用程序让用户可以用表格或图表的形式看到众多的数据, 而且用户可能想同时看到这两种形式的数据: 表格数据和根据该表格数据而绘制的图表。那么, 可以在分离的框架窗口或在一个窗口中通过切分窗口来实现。现在, 应用程序支持用户在表格中编辑数据, 并可以立即在图表里看到这些变化。在 MFC 类库中, 表格视图类和图表视图类是基于从 `CView` 类中派生出的不同视图类。两个视对象都可以和一个文档对象交互。该文档存储数据(也可能从一个数据库中获得数据)。两个视都可以访问到文档, 并显示从文档中获得的数据。当一个用户更新多视中的一个视时, 视类调用文档类的 `UpdateAllViews` 函数。这个函数通知所有在此文档中的视, 让每一个视都用文档中最近的数据更新自己。单独调用 `UpdateAllViews` 函数就可以使不同的视同步。要达到这样的效果, 没有数据和视的分离, 编码将是很困难的, 特别是在视自己存储数据时更是如此。使用文档/视图结构, 这将变得很容易。应用程序框架为程序员做了很多协调的工作。

MFC 类库使基于单文档界面(SDI)和多文档界面(MDI)的应用程序开发变得非常简单。在 Visual C++ 中, 运用 AppWizard 工具可以方便的生成这两种应用程序。当一个应用程序在 Windows 下运行时, 用户与窗口框架中显示的文档界面进行交互。一个文档窗口的框架有两个组成部分: 框架和它所包含的部分。一个文档窗口可以是一个单文档界面(SDI)的窗口, 也可以是一个多文档界面(MDI)的子窗口。Windows 管理着大多数用户与之交互的命令, 这些命令包括移动、缩放、关闭和最小化、最大化等。而用户一般只管理窗口框架中处理的消息。

1.3.2 文档和 `CDocument` 类

文档是由 MFC 类库中的 `CDocument` 类派生的对象, 当用户用 AppWizard 创建一个文档/视图结构的应用程序框架时, VC++ 会自动向应用程序中添加一个文档对象的框架结构。文档代表了用户存储或打开一个文件的一个数据单位。它的主要作用是把数据处理从界面处理中分离出来, 同时提供一个与其他对象交互的接口, 使每一种类都能相对独立的成为一个模块, 达到程序模块化设计的要求。

一个应用程序可以支持多种类型的文档。在应用程序中对其所需要支持的每一种类型的文档都必须创建一个相关的文档模板,这种文档模板描述了使用的文档资源。每一种文档包含了一个指向相关的 CDocTemplate 对象的指针,文档模板描述了每一种文档的视和窗口的风格。当用户打开一个文档时,应用程序框架就会创建一个与之相连的视图,一个文档可以有多个视图。用户通过与相应的视图与文档进行交互,视图类在应用程序框架中提供了文档类的一个映像,并把用户的输入解释为对文档的操作。文档作为程序框架命令循环的一部分,能够接受菜单、工具栏等界面元素发送的命令,如 File|Open 等。文档只接收活动的视中的命令,如果文档中没有处理一个给定命令的函数,它将会把命令传递到管理它的文档模板进行处理。如果文档中的数据被修改,那么它的每一个视都必须反映这些修改,而程序框架也会在文件关闭时提醒用户保存对文档的修改。

注意 文档类不能处理标准的 Windows 消息,只能处理 WM_COMMAND 消息。

CDocument 类为用户自定义的文档类提供了基类,下面给出在应用程序中使用文档类的典型步骤:

- (1) 以 CDocument 类为基类派生出自己的文档类。
- (2) 为在文档中为需要处理的数据添加成员变量。
- (3) 编写用于读取和修改文档数据的成员函数。

在 CDocument 类中还封装了许多对文档进行操作的成员函数,下面介绍一些主要的成员函数:

- UpdateAllViews 函数

调用此成员函数以便在视图中更新文档数据。该函数的原型如下:

```
void UpdateAllViews(CView* pSender, LPARAM lHint = 0L, CObject* pHint = NULL);
```

其中 pSender 是需要更新的视图指针,如果为 NULL 则更新所有视图,而 lHint 和 pHint 为提示参数,lHint 包含与修改有关的信息,可以传递任何数据;而 pHint 则指向保存修改信息的对象,可以传递从 CObject 类派生的任何对象的指针。利用这两个参数可以为视图提供高级应用,例如决定视图的哪些部分应该更新。

- OnNewDocument 函数

调用此成员函数以新建文档,该函数默认调用 DeleteContents 成员函数,以便清除当前文档对象中的所有数据。所有 CDocument 类的派生类都应该重载这个函数,以便清除用户添加的数据。如果用户在一个 SDI 应用程序中选择了 File|New 命令,应用程序框架将用这个函数重新初始化已经存在的文档,而不是重新再创建一个新文档。如果用户在一个 MDI 应用程序中选择了 File|New 命令,应用程序框架将创建一个新的文档,并调用这个函数以初始化这个文档。

注意 只有将初始化代码添加在这个函数而不是在构造函数中,才能使 SDI 应用程序中的 File|New 命令产生作用。

- OnOpenDocument 函数

调用此成员函数打开特定的文件。该函数首先调用 DeleteContents 成员函数清空文

档,然后调用 `CObject::Serialize` 函数读取文件的内容。如果用户在 SDI 应用程序中选择了菜单中的 `File|Open` 命令,应用程序框架将会用这个函数加载文档,而不是创建一个新文档。如果用户在 MDI 应用程序中选择了 `File|Open` 命令,应用程序框架将构造一个新的 `CDocument` 对象,并调用这个函数去初始化它。

注意 必须把自己的初始化代码放置在这个函数里,而不是构造函数里,以使 SDI 应用程序菜单中的 `File|Open` 命令发生作用。

1.3.3 视图与 CView 类

视图是主窗口下的子窗口。多个视可以共享一个主窗口,`CDocTemplate` 为视图、主窗口和文档之间创建了相互联系。当用户打开一个新的窗口或切分一个已经存在的窗口时,应用程序将构造一个新的视图,并将其连接到相应的文档。一个视图只能对应一个文档,而一个文档则可以同时对应多个视图,例如 Word 可以同时打开文本编辑视图和文档结构视图。

视图可以处理多种类型的输入和命令,例如键盘输入、鼠标输入、菜单命令、工具栏命令等。视图接受由程序框架传递给它的命令,如果视图不能处理该命令,就将该命令传递给对应的文档来处理。

视图负责显示和修改文档数据,在应用程序中视图可以直接读取文档的数据,也可以在文档中提供由视图调用的函数来获取文档数据。当文档中的数据发生变化时,视图一般通过调用 `CDocument::UpdateAllViews` 函数来显示这些变化。在应用程序中使用视图,首先从 `CView` 基类派生出自己的视图类,然后重载 `OnDraw` 函数来进行屏幕显示、打印和打印预览。

除了 `Cview` 基类外,MFC 类库还提供了几个专用的由 `CView` 类派生的视图类,以实现特定的功能:

- `CScrollView` 类: 提供滚动支持与缩放显示功能的视图。
- `CFormView` 类: 提供以对话框资源为模板的可滚动视图。
- `CRecordView` 和 `CDaoRecordView` 类: 提供使用以对话框控件显示数据库记录的视图。
- `CEditView` 类: 提供能够使用简单的多行文本编辑控件的视图。
- `CctrlView` 类: 提供能够使用树、列表和编辑控件的视图。
- `CListView` 类: 提供能够使用列表控件的视图。
- `CRichEditView` 类: 提供能够使用复杂编辑控件的视图。
- `CTreeView` 类: 提供能够使用树控件的视图。

`CView` 类为用户自定义的视图类提供了基类,在 `CView` 类中还封装了许多对视图进行操作的成员函数,这些函数能够处理显示、获取数据以及初始化数据等操作。下面是 `CView` 类的一些重要函数:

- `OnDraw` 函数: 调用该成员函数进行屏幕显示、打印和打印预览。`OnDraw` 函数通过调用文档数据成员函数以获得文档数据,并将其传递给设备环境对象的成员函数以显示文档数据。
- `GetDocument` 函数: 调用该成员函数以获得一个指向当前视所对应的文档类指针。

前面提到过,视图只与一个文档对象相联系,而 GetDocument 函数允许应用程序在视图中获得对应的文档指针。当用户在视图中输入新的数据时,视图必须通知文档对象对其内部数据进行更新,而使用 GetDocument 函数得到指向文档的指针,利用它就能够对文档类的成员函数和公共数据成员进行访问。GetDocument 函数的原型如下:

```
CDocument * GetDocument() const;
```

该函数返回指向与视图对应的文档对象指针,如果没有文档与视图对应,则返回 NULL。

在由 AppWizard 生成的 CView 派生类中,AppWizard 会添加一个保证文档类型的 GetDocument() 函数,该函数返回的是指向派生文档类而不是 CDocument 的指针,其定义类似于如下形式:

```
CMyDoc * GetDocument(){return (CMyDoc *) m_pDocument;};
```

其中 CMyDoc 为派生文档类。

- OnUpdate 函数:调用该成员函数以使视图反应文档数据的变化,该函数被文档类中的 UpdateAllViews 函数所调用。不过用户也可以直接在派生视图类中调用该函数。通常派生视图类的 OnUpdate 函数要访问文档,读取文档的数据,然后再更新视图的数据成员或控件,以便反映出文档的变化。默认情况下,OnUpdate 使整个视图窗口无效,同时触发 OnDraw 函数,使用更新后的文档数据重新绘制窗口。
- OnInitialUpdate 函数:框架窗口在视图第一次显示时调用该函数。在基类的 OnInitialUpdate 函数中,除了调用 OnUpdate 函数外,不做其他任何事情。如果需要在派生视图类中重载 OnInitialUpdate 函数,则一定要在其中调用基类的 OnInitialUpdate 函数,或者调用派生视类的 OnUpdate 函数。通过重载 OnInitialUpdate 函数可以完成视类的初始化。应用程序框架在创建视图时,首先调用 OnCreate 函数,然后调用 OnInitialUpdate 函数。OnCreate 函数只能够被调用一次,而 OnInitialUpdate 函数可以被调用多次。
- 打印函数:视图类所提供的打印函数如表 1-4 所示。

表 1-4 CView 类的打印函数

打印函数	函数功能
OnBeginPrinting	重载此函数以创建打印文档时所需要的资源,在此函数中也能够设置最大打印页数
OnEndPrinting	重载此函数以释放在 OnBeginPrinting 函数中创建的资源
OnPreParePrinting	重载此函数以提供文档的最大打印页数
OnPrePareDC	重载此函数来修改用于显示或打印文档的设备环境
OnPaint	重载此函数以提供如打印页眉、页脚等的附加打印服务

1.3.4 框架窗口

框架窗口亦即应用程序的主窗口,是由窗口类管理的。SDI 应用程序框架窗口所继承的类为 `CFrameWnd` 类;而 MDI 应用程序的框架窗口所继承的类为 `CMDIFrameWnd` 类,其文档框架窗口所继承的类为 `CMDIChildWnd` 类。每个文档都有一个文档框架窗口,而一个文档框架窗口至少含有一个视图以显示文档数据。

SDI 应用程序的框架窗口既是主框架窗口又是文档框架窗口,其框架窗口在应用程序运行时创建,当应用程序结束时销毁。而 MDI 应用程序的主框架窗口随应用程序的运行而创建,随应用程序的结束而销毁;其文档框架窗口则当打开或新建文档时创建,而在文档关闭时销毁。

当使用 AppWizard 创建文档/视图应用程序时,AppWizard 会自动为应用程序的主框架窗口派生一个框架窗口类,默认情况下为 `CMainFrame` 类。对于 MDI 应用程序,其主框架窗口管理菜单栏、标题栏、工具栏以及其他控件。

本章小结

本章主要向读者介绍了 Visual C++ 用户界面设计基础、Windows 编程机制和 Windows 应用程序的结构。通过本章的学习,读者应该达到以下几点:

- 掌握界面设计的一般原则。
- 理解 Windows 基于消息的运行机制。
- 了解 MFC 类库的结构。
- 理解 MFC 应用程序的结构。

第2章 按钮控件

按钮控件是 Windows 应用程序中最为常见的控件之一,本章主要向读者介绍如何设计与众不同的按钮。

本章要点:

- CButton 类的使用;
- 定制不同类型的按钮。

2.1 按钮控件编程基础

按钮控件是一个小的矩形子窗口,可以被单击。按钮能够单独使用或成组使用,可以有标题也可以没有标题。通常当用户单击按钮时,它们的外观会发生变化。

在 MFC 中,Windows 按钮控件是由 CButton 类提供支持的,该类的派生结构如图 2-1 所示。典型的按钮包括复选框、单选按钮和下压式按钮。CButton 对象可以是其中的任何一种按钮,这由该对象 Create 成员函数的按钮类型参数指定。

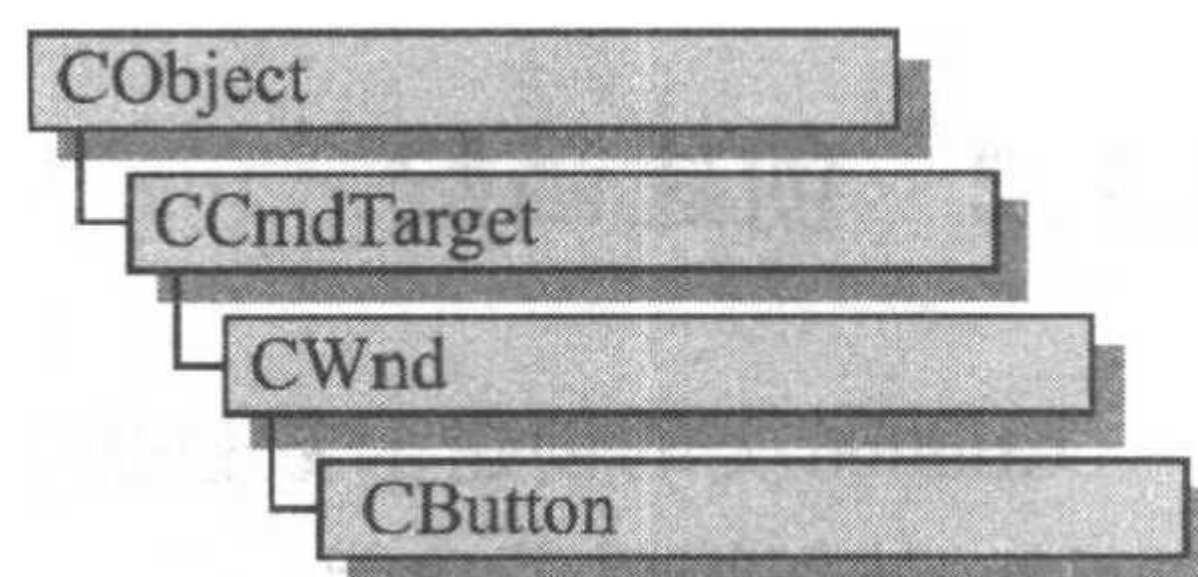


图 2-1 CButton 类的派生结构

另外,CButton 类的派生类 CBitmapButton 类支持创建以位图作为按钮控件的标题。对于按钮控件的弹起、按下和聚焦状态,CBitmapButton 对象能使用不同的位图图像。

2.1.1 按钮控件概述

用户既可以在对话框模板中创建按钮控件,也可以在代码中直接创建。在这两种情况下,都需要首先调用构造函数 CButton 以创建 CButton 对象,接着调用 Create 成员函数以创建 Windows 按钮控件,并将其与 CButton 对象连接。

用 CButton 派生类创建按钮可以一步实现。在派生类中设计一个构造函数,并在其中直接调用 Create 函数。

如果希望处理由按钮控件向其父窗口(通常是对话框)发送的 Windows 通告消息,则需要在相应的父窗口类中添加消息映射入口和消息处理函数。

每个消息映射入口都具有以下形式：

```
ON_Notification(id, memberFxn)
```

其中 id 指定了发送通告消息的控件 ID,而 memberFxn 则指定了用于处理控件通告消息的成员函数。

而消息处理函数如下：

```
afx_msg void memberFxn( );
```

可以处理的消息映射项及其响应的操作如表 2-1 所示。

表 2-1 消息映射及对应操作

映射项	操作
ON_BN_CLICKED	用户单击按钮
ON_BN_DOUBLECLICKED	用户双击按钮

如果从对话框资源中创建 CButton 对象,则对话框关闭时,CButton 对象会自动销毁。

如果在窗口中创建 CButton 对象,则需要自己销毁它。如果创建使用 new 函数在堆中创建,则在用户关闭 Windows 按钮控件时,需要对该对象调用 delete 函数。如果在堆栈中创建 CButton 对象,或将其嵌入父对话框对象中,它会自动被销毁。

2.1.2 创建函数

CButton 类的创建函数包括 CButton 和 Create,它们可以完成构造 CButton 对象,创建 Windows 按钮控件等功能。

- CButton

调用该函数以构造一个 CButton 对象,其原型为：

```
CButton( );
```

- Create

调用该函数以创建 Windows 按钮控件,并将其与 CButton 对象相联系,其原型为：

```
BOOL Create( LPCTSTR lpszCaption, DWORD dwStyle, const RECT& rect, CWnd * pParentWnd, UINT nID );
```

返回值：

如果函数调用成功则返回非零值,否则返回零值。

参数：

lpszCaption —— 指定了按钮控件标题,例如“确定”按钮上的“确定”就是控件标题。

dwStyle —— 指定了按钮控件的风格,其取值可以为表 2-2 中值的任意组合。

表 2-2 dwStyle 取值

dwStyle 取值	含义
BS_AUTOCHECKBOX	创建的按钮与复选框相同,只是当用户选中该按钮时,检查标记会出现在复选框中;而当用户下一次选中该按钮时,检查标记会消失
BS_AUTORADIOBUTTON	创建的按钮与单选按钮相同,只是当用户选中该按钮时,按钮会自动高亮显示;而当用户选中同组中其他按钮时,该按钮恢复正常显示
BS_AUTO3STATE	创建的按钮与三态复选框相同,只是复选框会在用户选中时改变状态
BS_CHECKBOX	创建一个小正方形,在其右边显示文本,除非同时指定 BS_LEFTTEXT 风格
BS_DEFPUSHBUTTON	创建具有黑边框的按钮,用户可以通过按下 Enter 键选择该按钮。这种风格使用户能够迅速选择默认选项
BS_GROUPBOX	创建组框,所有与该风格有关的文本都将显示在矩形的左上角
BS_LEFTTEXT	当与复选框或单选按钮风格共同使用时,文本显示在单选按钮或复选框的左边
BS_OWNERDRAW	创建自绘制按钮,当按钮的某个可见部分发生变化时,框架将调用 DrawItem 成员函数。当使用 CBitmapButton 类时,必须指定该风格
BS_PUSHBUTTON	创建下压式按钮,当用户按下该按钮时,它会向其父窗口发送 WM_COMMAND 消息
BS_RADIOBUTTON	创建在其右边显示文本的小圆圈(除非与 BS_LEFTTEXT 风格共同使用),单选按钮通常成组使用
BS_3STATE	创建的按钮与复选框相同,只是其状态除了选中和未选外,还可以有灰色态(第三态)。灰色态通常表示复选框被禁止

rect —— 指定了按钮控件的尺寸和位置,该参数可以为 CRect 对象,也可以为 RECT 结构。

pParentWnd —— 指定了按钮控件的父窗口,通常为对话框。该参数不能为 NULL。

nID —— 指定了按钮控件的 ID。

创建 CButton 对象通常需要两个步骤:首先调用构造函数,然后调用 Create 成员函数将所构造的 CButton 对象与 Windows 按钮控件相联系。

按钮控件能够使用的窗口风格常数如表 2-3 所示。

表 2-3 能够对按钮控件使用的窗口风格常数

风格常数	含义
WS_CHILD	子窗口
WS_VISIBLE	窗口可见
WS_DISABLED	窗口被禁止
WS_GROUP	成组按钮
WS_TABSTOP	使按钮能够用 Tab 键切换

如果指定了 WS_VISIBLE 风格,则 Windows 会向按钮控件发送所有按钮激活或屏蔽

需要的消息。

2.1.3 操作函数

CButton 类的操作函数包括：GetState、SetState、GetCheck、SetCheck、GetButtonStyle、SetButtonStyle、GetIcon、SetIcon、GetBitmap、SetBitmap、GetCursor 和 SetCursor 函数，它们可以完成获得按钮状态、设置按钮状态等功能。

• GetState

调用该函数以得到按钮控件的选中状态、高亮状态或焦点状态，其原型为：

```
UINT GetState( ) const;
```

返回值：

得到按钮控件的当前状态，用户可以将表 2-4 中给出的状态掩码与返回值执行与操作，以从返回值中提取出状态信息。

表 2-4 状态掩码

状态掩码	含义
0x0003	使用该状态掩码可以得到选中状态(只对单选按钮和复选框有效),所得结果为 0 表示按钮未选中;结果为 1 则表示按钮被选中。当单选按钮被选中时,其中包含圆点;而当复选框被选中时,其中包含一个“X”。当所得结果为 2 时,表示选中状态未定(只有三态复选框能得到该结果)。当三态复选框为灰色时,它的状态未定
0x0004	使用该状态掩码可以得到高亮状态。如果所得结果为非零值,则表示按钮处于高亮状态。当用户单击按钮,同时按下鼠标左键不放时,按钮处于高亮状态;当用户释放鼠标左键时,高亮态被去除
0x0008	使用该状态掩码可以得到焦点状态。如果所得结果为非零值,则按钮具有输入焦点

• SetState

调用该函数以设置按钮控件的高亮态，其原型为：

```
void SetState( BOOL bHighlight );
```

参数：

bHighlight —— 指定了是否将按钮设置为高亮态。如果该参数为非零值，则按钮被设置为高亮；如果该参数为零值，则按钮的高亮态被去除。

高亮态会使按钮控件的外观发生变化，但不会影响单选按钮或复选框的选中状态。当用户单击按钮，同时按下鼠标左键不放时，按钮处于高亮状态；而当用户释放鼠标左键时，高亮态被去除。

• GetCheck

调用该函数以得到按钮控件的选中状态，其原型为：

```
int GetCheck( ) const;
```

返回值：

如果按钮控件在创建时指定了 BS_AUTOCHECKBOX、BS_AUTORADIOBUTTON、BS_AUTO3STATE、BS_CHECKBOX、BS_RADIOBUTTON 或 BS_3STATE 风格,则返回值如表 2-5 中给出值之一。

表 2-5 返回值

返回值	含义
0	按钮处于非选中态
1	按钮处于选中态
2	按钮状态不能确定

如果按钮具有其他风格,则返回值为 0。

• SetCheck

调用该函数以设置按钮控件的选中状态,其原型为：

```
void SetCheck( int nCheck );
```

参数：

nCheck —— 指定了按钮控件的选中状态,其取值可以为表 2-6 中给出值之一。

表 2-6 nCheck 参数取值

nCheck 取值	含义
0	设置按钮状态为未选中态
1	设置按钮状态为选中态
2	设置按钮为不确定态,该值只对具有 BS_3STATE 或 BS_AUTO3STATE 风格的按钮控件有效

SetCheck 函数能够设置单选按钮或复选框的状态,但对下压式按钮无效。

• GetButtonStyle

调用该函数以得到按钮控件风格信息,其原型为：

```
UINT GetButtonStyle( ) const;
```

返回值：

返回 CButton 对象的按钮风格常数,具体参见表 3-2。

• SetButtonStyle

调用该函数以改变按钮的风格,其原型为：

```
void SetButtonStyle( UINT nStyle, BOOL bRedraw = TRUE );
```

参数：

nStyle —— 指定了按钮风格,取值参见表 3-2。

bRedraw —— 指定了按钮是否需要被重绘。如果该参数为非零值,则按钮需要重绘；如果该参数为 0 值,则无需重绘按钮。默认情况下,按钮将被重绘。

调用 `GetButtonStyle` 成员函数可以得到按钮的风格,其低位字为按钮风格。

- **GetIcons**

调用该函数以得到由 `SetIcon` 函数设置的图标句柄,其原型为:

```
HICON GetIcon( ) const;
```

返回值:

如果函数调用成功,则返回与按钮控件相联系的图标句柄。如果先前没有指定图标,则返回 `NULL`。

- **SetIcon**

调用该函数以指定将在按钮上显示的图标,其原型为:

```
HICON SetIcon( HICON hIcon );
```

返回值:

如果函数调用成功,则返回先前与按钮相联系的图标句柄。

参数:

`hIcon` —— 指定了将与按钮相联系的图标句柄。

`SetIcon` 函数将指定图标与按钮相联系。图标将被自动显示在按钮表面,默认情况下居中显示。如果图标太大,则图标的两侧都将被从按钮边缘处截断。当然,用户也可以指定其他对齐方式,例如: `BS_TOP`(顶对齐)、`BS_LEFT`(左对齐)、`BS_RIGHT`(右对齐)、`BS_CENTER`(水平居中)、`BS_BOTTOM`(底对齐)和 `BS_VCENTER`(垂直居中)。

`SetIcon` 只为按钮分配一个图标,这与 `CBitmapButton` 类不同(每个按钮使用4个位图)。当按钮被按下时,图标将向右下方平移,以表现出下压的效果。

- **GetBitmap**

调用该函数以得到由 `SetBitmap` 函数设置的位图句柄,其原型为:

```
HBITMAP GetBitmap( ) const;
```

返回值:

如果函数调用成功,则返回位图句柄;如果先前未指定位图,则返回 `NULL`。

- **SetBitmap**

调用该函数以指定将在按钮上显示的位图,其原型为:

```
HBITMAP SetBitmap( HBITMAP hBitmap );
```

返回值:

如果函数调用成功,则返回先前与按钮相联系的位图句柄。

参数:

`hBitmap` —— 指定了将与按钮相联系的位图句柄。

`SetBitmap` 函数将指定位图与按钮相联系。位图将被自动显示在按钮表面,默认情况下居中显示。如果位图太大,则位图的两侧都将被从按钮边缘处截断。当然,用户也可以指定其他对齐方式,例如: `BS_TOP`(顶对齐)、`BS_LEFT`(左对齐)、`BS_RIGHT`(右对齐)、

BS_CENTER(水平居中)、BS_BOTTOM(底对齐)和 BS_VCENTER (垂直居中)。

SetBitmap 只为按钮分配一个位图,这与 CBitmapButton 类不同(每个按钮使用 4 个位图)。当按钮被按下时,位图将向右下方平移,以表现出下压的效果。

- GetCursor

调用该函数以得到由 SetCursor 函数设置的光标句柄,其原型为:

```
HCURSOR GetCursor( );
```

返回值:

如果函数调用成功,则返回光标图像句柄;如果先前未指定光标,则返回 NULL。

- SetCursor

调用该函数以指定将在按钮上显示的光标图像,其原型为:

```
HCURSOR SetCursor( HCURSOR hCursor );
```

返回值:

如果函数调用成功,则返回先前与按钮相联系的位图句柄。

参数:

hCursor —— 指定了将与按钮相联系的光标句柄。

SetCursor 函数将指定光标图像与按钮相联系。光标图像将被自动显示在按钮表面,默认情况下居中显示。如果光标图像太大,则光标图像的两侧都将被从按钮边缘处截断。当然,用户也可以指定其他对齐方式,例如: BS_TOP(顶对齐)、BS_LEFT(左对齐)、BS_RIGHT(右对齐)、BS_CENTER(水平居中)、BS_BOTTOM(底对齐)和 BS_VCENTER (垂直居中)。

SetCursor 只为按钮分配一个光标,这与 CBitmapButton 类不同(每个按钮使用 4 个位图)。当按钮被按下时,位图将向右下方平移,以表现出下压的效果。

2.1.4 重载函数

CButton 类提供了重载函数 DrawItem,它用于自己绘制按钮控件,其原型为:

```
virtual void DrawItem( LPDRAWITEMSTRUCT lpDrawItemStruct );
```

参数:

lpDrawItemStruct —— 为指向 DRAWITEMSTRUCT 结构的指针,它为需要自己绘制的对象(控件或菜单项)提供了必要的信息,它决定了绘制的方式和细节。需要自己绘制的控件或菜单项的父窗口,将指向该结构的指针作为 WM_DRAWITEM 消息的 lParam 参数。DRAWITEMSTRUCT 结构的定义如下:

```
typedef struct tagDRAWITEMSTRUCT {  
    UINT CtlType;  
    UINT CtlID;  
    UINT itemID;
```

```
    UINT  itemAction;  
    UINT  itemState;  
    HWND  hwndItem;  
    HDC   hdc;  
    RECT  rcItem;  
    DWORD itemData;  
};  
DRAWITEMSTRUCT;
```

注意 DRAWITEMSTRUCT 结构并非只是针对按钮控件,相反它是针对所有自绘制对象的。

结构成员:
CtlType —— 指定了控件类型,其取值如表 2-7 所示。

表 2-7 CtlType 成员取值

CtlType 成员取值	含义
ODT_BUTTON	自绘制按钮
ODT_COMBOBOX	自绘制组合框
ODT_LISTBOX	自绘制列表框
ODT_MENU	自绘制菜单项
ODT_LISTVIEW	自绘制列表视
ODT_STATIC	自绘制静态控件
ODT_TAB	Tab 控件

CtlID —— 指定了需要自己绘制的控件 ID,而对于菜单项则无需使用该成员。

itemID —— 可以为菜单项 ID、列表框或组合框中某项的索引。对于空列表框或组合框,该成员为负值,这时应用程序只绘制焦点矩形(其坐标由 rcItem 成员给出)。虽然此时控件中没有需要显示的项,但绘制焦点矩形还是很有必要的,因为这能够提示用户该控件是否具有输入焦点。当然,也可以设置 itemAction 成员为合适的值,使得无需绘制输入焦点。

itemAction —— 指定了绘制行为,其取值可以为表 2-8 中所示值的一个或几个的联合。

表 2-8 itemAction 成员取值

itemAction 成员取值	含义
ODA_DRAWENTIRE	当整个控件都需要被绘制时,设置该值
ODA_FOCUS	如果控件需要在获得或失去输入焦点时被绘制,则设置该值。此时应该检查 itemState 成员,以确定控件是否具有输入焦点
ODA_SELECT	如果控件需要在选中状态改变时被绘制,则设置该值。此时应该检查 itemState 成员,以确定控件是否处于选中状态

itemState —— 指定了当前绘制操作完成后,所绘项的可见状态。例如,如果菜单项应该被灰色显示,则可以指定 ODS_GRAYED 状态标志。itemState 成员的取值如表 2-9 所示。

表 2-9 itemState 成员取值

itemState 成员取值	含义
ODS_CHECKED	如果菜单项将被选中,则设置该值(只对菜单项有效)
ODS_DISABLED	如果绘制项将被禁止,则设置该值
ODS_FOCUS	如果绘制项需要输入焦点,则设置该值
ODS_GRAYED	如果绘制项需要被灰色显示,则设置该值
ODS_SELECTED	如果绘制项需要被设置为选中状态,则设置该值
ODS_COMBOBOXEDIT	在自绘制组合框中只绘制选择区域(编辑控件中)
ODS_DEFAULT	自绘制项为默认项

hwndItem —— 指定了组合框、列表框和按钮等自绘制控件的窗口句柄;或包含菜单项的菜单句柄(HMENU)。

hDC —— 指定了绘制操作所用的设备环境。

rcItem —— 指定了将被绘制的矩形区域。Windows 将自动地裁剪组合框、列表框或按钮等控件的自绘制区域外部分。由于 Windows 不裁剪菜单项,因此在绘制菜单项时,必须保证绘制操作在 rcItem 成员所指定的区域中进行。

itemData —— 对于列表框或组合框,该成员的取值可以由 CComboBox::AddString、CComboBox::InsertString、CListBox::AddString 或 CListBox::InsertString 等传递给控件的值。

对于菜单项,该成员取值可以由 CMenu::AppendMenu、CMenu::InsertMenu 或 CMenu::ModifyMenu 等传递给菜单的值。

对于按钮来说,当其任何可视部分发生变化时,框架将调用 DrawItem 函数进行绘制。自绘制按钮具有 BS_OWNERDRAW 风格。通过重载 DrawItem 函数,用户可以定制绘制内容和方式。在成员函数结束前,应用程序应该恢复先前的 GDI 对象。这一点非常重要,否则有可能出现异常情况。

2.1.5 CBitmapButton 类

CBitmapButton 类是 CButton 类的派生类,其派生结构如图 2-2 所示。该类专门用于管理以位图作为标签的下压式按钮控件(即所谓的位图按钮)。CBitmapButton 对象包括 4 张位图,分别表示按钮的弹起态、下压态、焦点态和禁止态。只有第一个位图是必需的,其他三张都是可选的。

位图按钮的图像也包括其四周的边界,而边界通常用于表示按钮的不同状态。例如,焦点态下的按钮其图像边界通常会被加上粗实线。这些位图可以为任意尺寸,但是最终都是按照弹起态位图的尺寸进行处理。

创建位图按钮时,首先要使按钮具有 BS_OWNERDRAW 风格。这样 Windows 会向按钮发送 WM_MEASUREITEM 和 WM_DRAWITEM 消息;而框架将负责处理这些消息以显示不同的位图。在 Windows 客户区中创建位图按钮

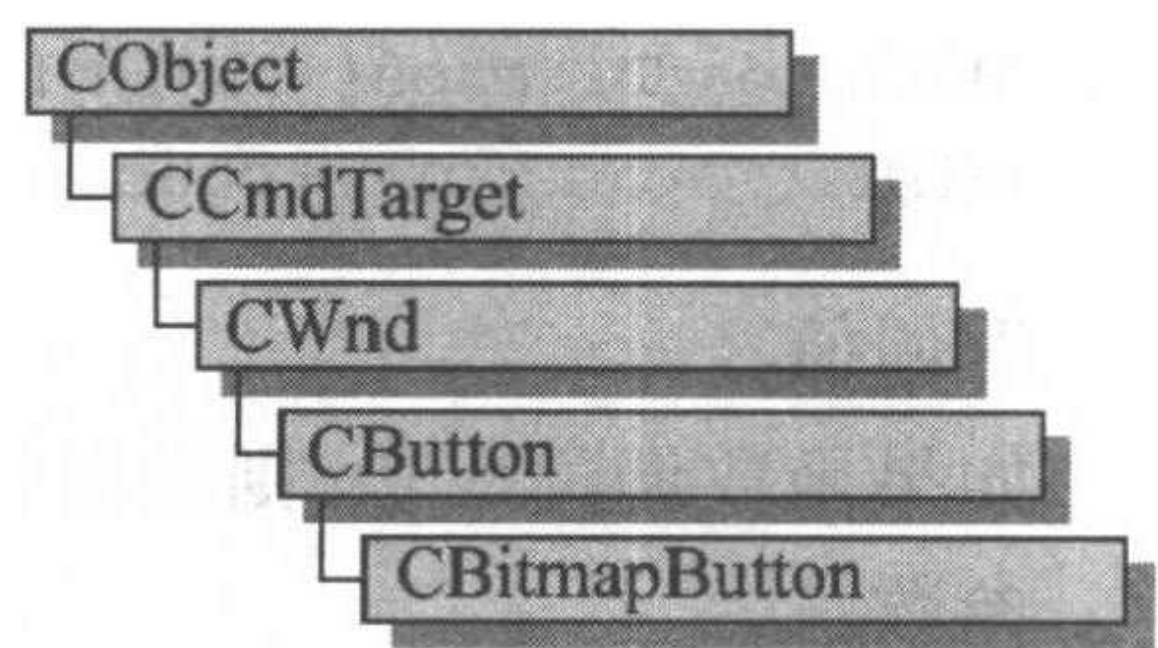


图 2-2 CBitmapButton 类的派生结构

(保护 4 个位图图像)的一般步骤如下:

(1) 构造 CBitmapButton 对象;

(2) 调用 Create 成员函数以创建 Windows 按钮控件,并将其与 CBitmapButton 对象关联;

(3) 调用 LoadBitmap 成员函数载入位图资源。

在对话框中包括位图按钮控件的一般步骤如下:

(1) 为按钮创建 1 到 4 个位图图像。

(2) 创建对话框模板,并在其中放置一个具有自绘制风格的按钮,按钮的尺寸没有影响。

(3) 如果将按钮的标签设置为“MYIMAGE”,则应该将按钮的 ID 设置为 IDC_MYIMAGE。

(4) 在应用程序的资源脚本中,将按钮的图片 ID 后分别添加一个字母: U、D、F 或 X (分别对应与弹起态、下压态、焦点态和禁止态)。例如,如果按钮标签为“MYIMAGE”,则图片的 ID 应该为“MYIMAGEU”、“MYIMAGED”、“MYIMAGEF”和“MYIMAGEX”。记住,位图 ID 中必须包含双引号,否则资源编辑器将为其赋予一个整数值,这样 MFC 将不能为按钮加载图片。

(5) 在应用程序对话框管理类中添加一个 CBitmapButton 对象。

(6) 在 OnInitDialog 函数中,调用 CBitmapButton 对象的 AutoLoad 函数。

(7) 有关 CBitmapButton 按钮的消息处理,与常规按钮控件一样。

CBitmapButton 类的成员函数有以下几个:

- CBitmapButton

调用该函数以构造一个 CBitmapButton 对象,其原型为:

```
CBitmapButton( );
```

- LoadBitmaps

调用该函数以载入位图,并将其与位图按钮对象相联系,其原型为:

```
BOOL LoadBitmaps( LPCTSTR lpszBitmapResource, LPCTSTR lpszBitmapResourceSel =  
NULL, LPCTSTR lpszBitmapResourceFocus = NULL, LPCTSTR lpszBitmapRe-  
sourceDisabled = NULL );  
BOOL LoadBitmaps( UINT nIDBitmapResource, UINT nIDBitmapResourceSel = 0, UINT  
nIDBitmapResourceFocus = 0, UINT nIDBitmapResourceDisabled = 0 );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

lpszBitmapResource —— 指定了按钮弹起态位图名,该参数不能为 NULL。

lpszBitmapResourceSel —— 指定了按钮下压态位图名,该参数可以为 NULL。

lpszBitmapResourceFocus —— 指定了按钮焦点态位图名,该参数可以为 NULL。

lpszBitmapResourceDisabled —— 指定了按钮禁止态位图名,该参数可以为 NULL。

nIDBitmapResource —— 指定了按钮弹起态位图 ID,该参数不能为 0。

nIDBitmapResourceSel —— 指定了按钮下压态位图 ID,该参数可以为 0。

nIDBitmapResourceFocus —— 指定了按钮焦点态位图 ID,该参数可以为 0。

nIDBitmapResourceDisabled —— 指定了按钮禁止态位图 ID,该参数可以为 0。

- AutoLoad

调用该函数以将对话框中的一个按钮与 CBitmapButton 类相联系,并根据按钮的名称和大小载入位图,其原型为:

```
BOOL AutoLoad( UINT nID, CWnd* pParent );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

nID —— 指定了按钮控件的 ID。

pParent —— 指定了按钮控件的父窗口。

2.2 改变按钮颜色

常规 Windows 按钮控件只能使用灰色,为什么我们不试着换一种颜色呢。例如,在制作一个 CD 播放器时,会大量地用到按钮。而此时使用灰色按钮,则显然和多媒体应用程序本身的基调不符。适当地改变颜色,无疑会为应用程序添加吸引力。如图 2-3 所示,即为一个彩色媒体播放器的界面。当然,为了使用更多的颜色作为示范,并没有顾及其颜色搭配。但是如果能够使用彩色按钮,并合适地搭配颜色,显然会使应用程序显得更加活泼。在本节中,就要向读者介绍彩色按钮的设计方法。

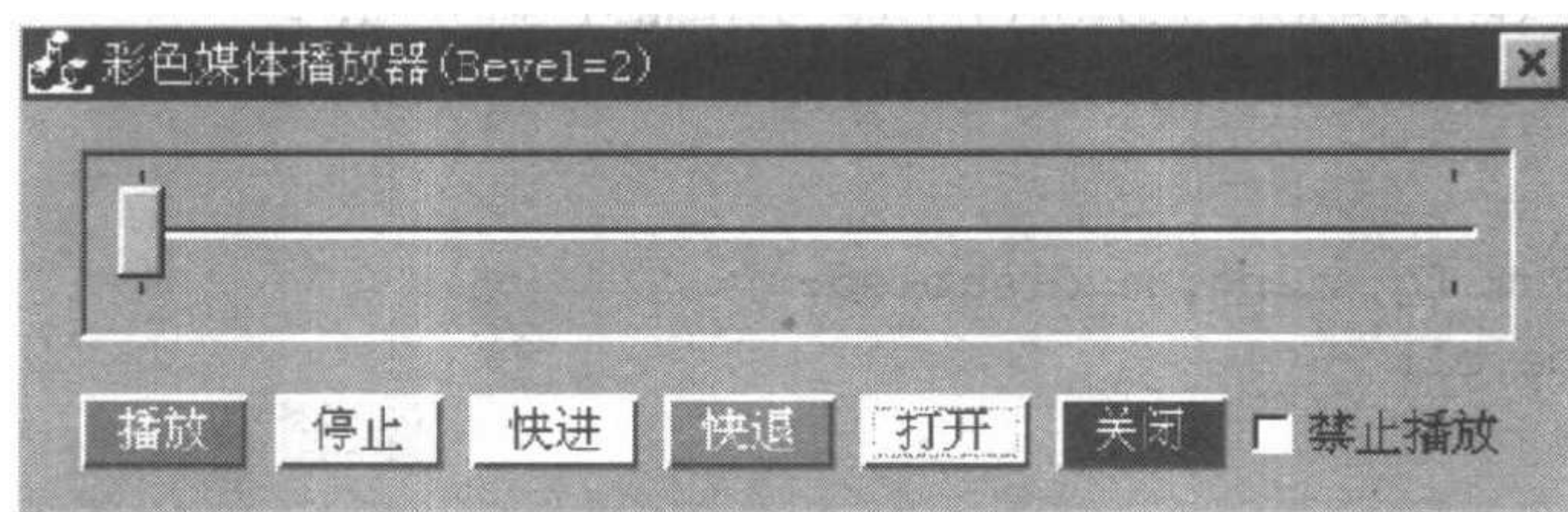


图 2-3 彩色媒体播放器

2.2.1 设计彩色按钮管理类

正如我们反复向读者强调的,如果超出 MFC 所提供的功能,就必须自己做一些工作。对于改变控件外观这一点来说,主要是利用自绘制功能(Owner-Draw)。下面将和读者一起设计一个 CColorButton 类,使用该类能够创建出彩色按钮。

首先,我们需要考虑一下 CColorButton 类应具有的功能。CColorButton 应该为 CButton

的派生类,这样它能够拥有 Windows 对按钮控件的常规支持。此外,它应该能够改变按钮控件、按钮标签以及按钮文本的颜色。

清单 2-1 所示为 CColorButton 类的定义:

清单 2-1 CColorButton 类定义

```
class CColorButton : public CButton
{
    DECLARE_DYNAMIC(CColorButton)
public:
    CColorButton();
    virtual ~CColorButton();

    BOOL Attach(const UINT nID, CWnd* pParent,
        const COLORREF BGColor = RGB(192, 192, 192),           // 灰色按钮
        const COLORREF FGColor = RGB(1, 1, 1),                 // 黑色文本
        const COLORREF DisabledColor = RGB(128, 128, 128),      // 按钮被禁止时文本也为
                                                                // 灰色
        const UINT nBevel = 2
    );

protected:
    virtual void DrawItem(LPDRAWITEMSTRUCT lpDIS);
    void DrawFrame(CDC * DC, CRect R, int Inset);
    void DrawFilledRect(CDC * DC, CRect R, COLORREF color);
    void DrawLine(CDC * DC, CRect EndPoints, COLORREF color);
    void DrawLine(CDC * DC, long left, long top, long right, long bottom,
        COLORREF color);
    void DrawButtonText(CDC * DC, CRect R, const char * Buf, COLORREF TextColor);

    COLORREF GetFGColor() { return m_fg; }
    COLORREF GetBGColor() { return m_bg; }
    COLORREF GetDisabledColor() { return m_disabled; }
    UINT GetBevel() { return m_bevel; }

private:
    COLORREF m_fg, m_bg, m_disabled;
    UINT m_bevel;
};
```

读者可以看到类中只提供了一个公有函数 Attach,它用以设置自绘制按钮的颜色。通常这个函数应该在对话框的 OnInitDialog 函数中调用。函数中 nID 参数为按钮控件的 ID;pParent 参数为父窗口指针,一般为 this;参数 BGColor 指定了按钮的背景色;参数 FGColor指定了按钮的前景色;参数 DisabledColor 指定了按钮禁止态颜色;参数 nBevel 指定了按钮的斜度。清单 2-2 所示为 Attach 函数的源代码:

清单 2-2 Attach()函数

```
BOOL CColorButton::Attach(const UINT nID, CWnd* pParent, const COLORREF
BGColor, const
COLORREF FGColor, const COLORREF DisabledColor, const UINT nBevel)
```

```

}

if (! SubclassDlgItem(nID, pParent))
    return FALSE;

m_fg = FGColor;
m_bg = BGColor;
m_disabled = DisabledColor;
m_bevel = nBevel;

return TRUE;
}

```

实际上,读者可能已经注意到这个函数的功能只是赋值。那么得到彩色按钮的工作是通过什么完成的呢?这就需要重载 CButton 基类的 DrawItem 函数,这个函数可以完成用户定制的绘制操作。清单 2-3 所示为 DrawItem 函数的清单:

清单 2-3 DrawItem()函数

```

void CColorButton::DrawItem(LPDRAWITEMSTRUCT lpDIS)
{
    CDC * pDC = CDC::FromHandle(lpDIS->hDC);

    UINT state = lpDIS->itemState;
    CRect focusRect, btnRect;
    focusRect.CopyRect(&lpDIS->rcItem);
    btnRect.CopyRect(&lpDIS->rcItem);
    //
    // 将焦点矩形的大小稍小于边界(一边减4)。
    //
    focusRect.left += 4;
    focusRect.right -= 4;
    focusRect.top += 4;
    focusRect.bottom -= 4;
    //
    //得到按钮标签
    //
    const int bufSize = 512;
    TCHAR buffer[bufSize];
    GetWindowText(buffer, bufSize);
    //
    // 使用定制的方法绘制彩色按钮
    //
    DrawFilledRect(pDC, btnRect, GetBGColor());
    DrawFrame(pDC, btnRect, GetBevel());
    DrawButtonText(pDC, btnRect, buffer, GetFGColor());
    //
    // 根据按钮的状态重新绘制按钮,例如绘制焦点矩形,或禁止按钮
    //
    if (state & ODS_FOCUS) {
        DrawFocusRect(lpDIS->hDC, (LPRECT)&focusRect);
        if (state & ODS_SELECTED) {

```



```

        DrawFilledRect(pDC, btnRect, GetBGColor());
        DrawFrame(pDC, btnRect, -1);
        DrawButtonText(pDC, btnRect, buffer, GetFGColor());
        DrawFocusRect(lpDIS -> hDC, (LPRECT)&focusRect);
    }
}
else if (state & ODS_DISABLED) {
    disabledColor = bg ^ 0xFFFFFFFF; // 反色
    DrawButtonText(pDC, btnRect, buffer, GetDisabledColor());
}
}

```

在重载函数中,首先得到按钮的位置和状态信息(从 LPDRAWITEMSTRUCT 结构的 rcItem 和 itemState 成员,具体参见 2.1 节)。然后设置焦点矩形的尺寸,并调用 GetWindowText 函数得到按钮的标签。这样就得到了绘制按钮所需的全部信息。接着调用 DrawFilledRect、DrawFrame 和 DrawButtonText 函数完成按钮的绘制。最后根据按钮的状态值绘制按钮。清单 2-4 所示为 DrawFilledRect 函数的源代码:

清单 2-4 DrawFilledRect() 函数

```

void CColorButton::DrawFilledRect(CDC * DC, CRect R, COLORREF color)
{
    CBrush B;
    B.CreateSolidBrush(color);
    DC -> FillRect(R, &B);
}

```

该函数的功能是绘制按钮的背景。

清单 2-5 所示为 DrawFrame 函数的源代码:

清单 2-5 DrawFrame() 函数

```

void CColorButton::DrawFrame(CDC * DC, CRect R, int Inset)
{
    COLORREF dark, light, tlColor, brColor;
    int i, m, width;
    width = (Inset < 0)? -Inset : Inset;

    for (i = 0; i < width; i += 1) {
        m = 255 / (i + 2);
        dark = PALETTERGB(m, m, m);
        m = 192 + (63 / (i + 1));
        light = PALETTERGB(m, m, m);

        if (width == 1) {
            light = RGB(255, 255, 255);
            dark = RGB(128, 128, 128);
        }

        if (Inset < 0) {
            tlColor = dark;
            brColor = light;
        }
    }
}

```

```

    }
    else {
        tlColor = light;
        brColor = dark;
    }

    DrawLine(DC, R.left, R.top, R.right, R.top, tlColor); // Across top
    DrawLine(DC, R.left, R.top, R.left, R.bottom, tlColor); // Down left

    if ( (Inset < 0) && (i == width - 1) && (width > 1) ) {
        // Across bottom
        DrawLine(DC, R.left + 1, R.bottom - 1, R.right, R.bottom - 1,
            RGB(1, 1, 1));
        // Down right
        DrawLine(DC, R.right - 1, R.top + 1, R.right - 1, R.bottom,
            RGB(1, 1, 1));
    }
    else {
        // Across bottom
        DrawLine(DC, R.left + 1, R.bottom - 1, R.right, R.bottom - 1,
            brColor);
        // Down right
        DrawLine(DC, R.right - 1, R.top + 1, R.right - 1, R.bottom,
            brColor);
    }
    InflateRect(R, -1, -1);
}
}

```

该函数的功能就是绘制按钮四周的斜坡。可以想像如果全部使用背景色绘制按钮,那么最后就没有立体效果。图 2-4 所示,即为不调用 DrawFrame 函数的界面效果。

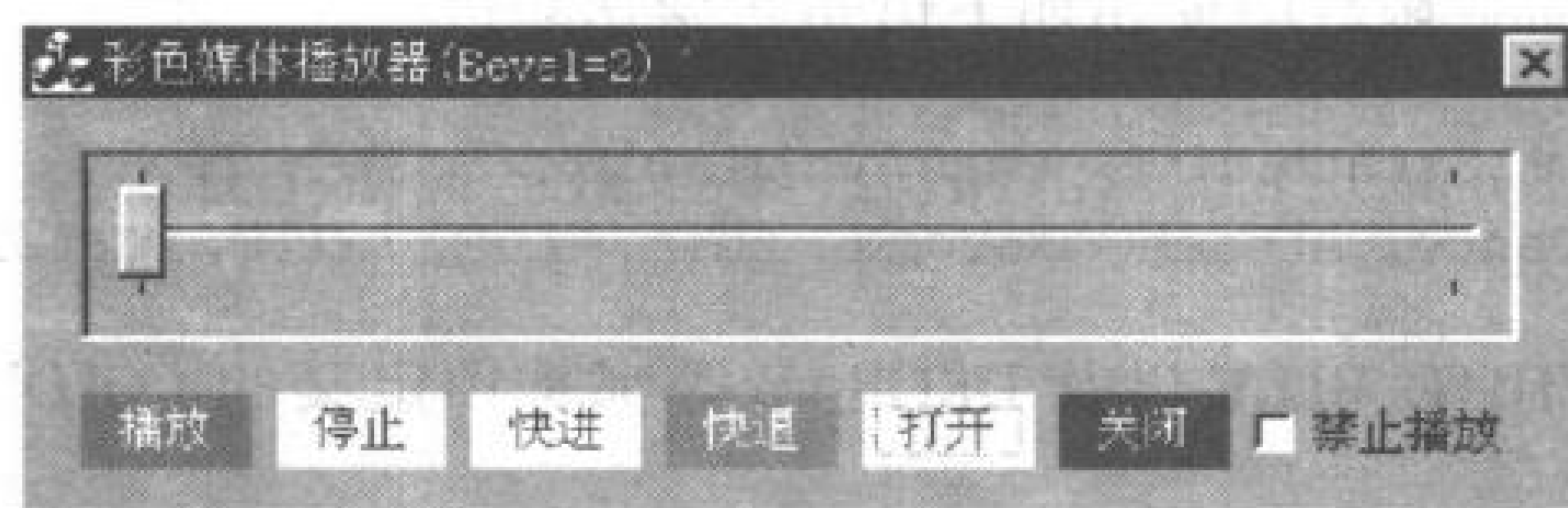


图 2-4 不调用 DrawFrame 函数的界面效果

这里顺便解释一下 m_bevel 成员,它主要定义了按钮边缘的坡度(立体效果)。图 2-5 所示为不同取值的 m_bevel 导致的不同绘制效果(其中左边按钮的 m_bevel 取值为 2,而右边 m_bevel 的取值为 4)。

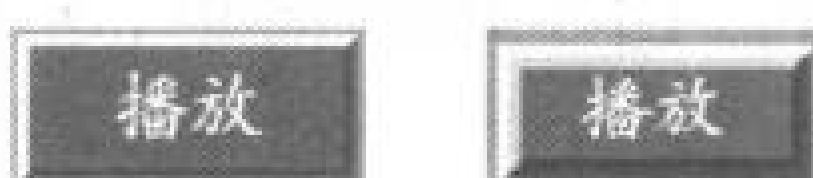


图 2-5 m_bevel 的作用

清单 2-6 所示为 DrawButtonText 函数的源代码：

清单 2-6 DrawButtonText() 函数

```
void CColorButton::DrawButtonText(CDC * DC, CRect R, const char * Buf, COLORREF
TextColor)
{
    COLORREF prevColor = DC->SetTextColor(TextColor);
    DC->SetBkMode(TRANSPARENT);
    DC->DrawText(Buf, strlen(Buf), R, DT_CENTER|DT_VCENTER|DT_SINGLELINE);
    DC->SetTextColor(prevColor);
}
```

该函数的功能是以指定的前景色绘制按钮标签。

类中的其他函数主要是用以得到成员取值或绘制直线之用,比较简单,这里就不再介绍了。

2.2.2 使用彩色按钮管理类

CColorButton 类的使用非常简单,步骤如下:

(1) 将 colorBtn.h 和 colorBtn.cpp 文件加入工程中;或如果已经将 CColorButton 类加入了 Gallery,则只需直接从其中插入即可。

(2) 在资源编辑器中创建按钮,并设置其属性为 Owner-Draw。

(3) 在应用程序的头文件中加入“# include colorbtn.h”,并以如下方式声明彩色按钮对象:

```
CColorButton m_btn1;
CColorButton m_btn2;
```

(4) 在 OnInitDialog 函数中使用如下语句初始化颜色:

```
VERIFY(m_btnOK.Attach(IDOK, this, CYAN, BLUE, DKCYAN));
VERIFY(m_btnCancel.Attach(IDCANCEL, this, BLUE, WHITE, DKBLUE));
```

其中 BLACK、WHITE、BLUE 以及 DKGRAY 等都是预定义的 COLORREF 常量,其定义类似如下形式:

```
const COLORREF CLOUDBLUE = RGB(128, 184, 223);
const COLORREF WHITE = RGB(255, 255, 255);
const COLORREF BLACK = RGB(1, 1, 1);
const COLORREF DKGRAY = RGB(128, 128, 128);
```

这样定义是为了使用方便,并且程序也更加直观易读。配套光盘中 chap3 \ colorbtn 目录为彩色媒体播放器的工程文件。由于比较简单,这里就不再多加解释,请读者自行参考源程序。

最后,还要向读者指出一些 CColorButton 类的待改进之处:

- 不支持动态 Create 函数直接创建。

- 不支持 256 色调色板。
- 不能改变按钮标签的字体。

2.3 改变按钮形状

上一节对按钮的颜色作出了改变,而本节将向读者介绍如何改变按钮的形状。

2.3.1 创建多边形按钮

这里以三角形按钮为例——从三角形很容易派生出多边形。首先设计一个三角形按钮管理类:CTriangleButton。该类是 CButton 类的派生类,并且如读者所预料的那样,主要功能是在重载的 DrawItem 函数中完成的。关于三角形的一个重要问题就是它的绘制方向(这是由于绘制三角形的算法需要下面还将提到)。因此类需要能够改变按钮的方向的功能,这些我们将一一在类中实现。清单 2-7 为 CTriangleButton 类的定义:

清单 2-7 CTriangleButton 类的定义

```
class CTriangleButton : public CButton
{
public:
    enum POINTDIRECTION {POINT_UP, POINT_DOWN, POINT_LEFT, POINT_RIGHT};
    // Construction
public:
    CTriangleButton();
    virtual ~CTriangleButton();

// Attributes
public:

protected:
    POINTDIRECTION PointDirection;
    CRgn CurrentRegion;

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{AFX_VIRTUAL(CTriangleButton)
public:
    virtual void DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct);
protected:
    virtual void PreSubclassWindow();
    //{AFX_VIRTUAL

public:
    BOOL SetWindowPos(const CWnd* pWndInsertAfter, int x, int y, int cx, int cy,
```



```

        UINT nFlags );
public:
    void SetDirection(POINTDIRECTION PointDirection);
    POINTDIRECTION GetDirection();
    // Generated message map functions
    protected:
    //{{AFX_MSG(CTriangleButton)
    //}}AFX_MSG

    DECLARE_MESSAGE_MAP()
};

```

POINTDIRECTION 结构中包含了 4 个方向信息,通过设置该参数就可以设置按钮的方向。默认的按钮方向为向右。

DrawItem 函数实现了类的最主要功能,其部分源代码(为了节省篇幅,类似部分都不在这里列出)如清单 2-8 所示:

清单 2-8 DrawItem()函数

```

void CTriangleButton::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
    ASSERT(lpDrawItemStruct != NULL);
    CRect rect = lpDrawItemStruct->rcItem;
    CDC * pDC = CDC::FromHandle(lpDrawItemStruct->hDC);
    UINT state = lpDrawItemStruct->itemState;
    UINT nStyle = GetStyle();

    int nSavedDC = pDC->SaveDC();

    rect.bottom = rect.right = min(rect.bottom, rect.right);
    pDC->FillSolidRect(rect, ::GetSysColor(COLOR_BTNFACE));

    rect.right -= 1; rect.bottom -= 1;

    CPen HighlightPen(PS_SOLID, 1, ::GetSysColor(COLOR_3DHIGHLIGHT));
    CPen DarkShadowPen(PS_SOLID, 1, ::GetSysColor(COLOR_3DDKSHADOW));
    CPen ShadowPen(PS_SOLID, 1, ::GetSysColor(COLOR_3DSHADOW));
    CPen BlackPen(PS_SOLID, 1, RGB(0,0,0));
    CPen FocusPen(PS_DOT, 0, RGB(0,0,0));

    // 绘制按钮
    switch (PointDirection) {
        case POINT_UP : {
            if (nStyle & BS_FLAT) { // 平面按钮
                pDC->SelectObject(BlackPen);
                pDC->MoveTo(rect.right / 2, 0);
                pDC->LineTo(0, rect.bottom);
                pDC->LineTo(rect.right, rect.bottom);
                pDC->LineTo(rect.right / 2, 0);
                pDC->SelectObject(HighlightPen);
                pDC->MoveTo(rect.right / 2, 2);
                pDC->LineTo(2, rect.bottom - 1);
            }
        }
    }
}

```

```

        pDC->LineTo(rect.right - 2, rect.bottom - 1);
        pDC->LineTo(rect.right / 2, 2);
    }
    else { //style not flat
        if ((state & ODS_SELECTED)) { // 按钮被按下
            pDC->SelectObject(HighlightPen);
            pDC->MoveTo(0, rect.bottom);
            pDC->LineTo(rect.right - 1, rect.bottom);
            pDC->LineTo(rect.right / 2, 0);

            pDC->SelectObject(ShadowPen);
            pDC->LineTo(0, rect.bottom);

            pDC->SelectObject(DarkShadowPen);
            pDC->MoveTo(rect.right / 2 - 1, 4);
            pDC->LineTo(1, rect.bottom);
        }
        else { //Button is not down
            pDC->SelectObject(HighlightPen);
            pDC->MoveTo(rect.right / 2, 0);
            pDC->LineTo(0, rect.bottom - 1);

            pDC->SelectObject(ShadowPen);
            pDC->LineTo(rect.right - 1, rect.bottom - 1);
            pDC->LineTo(rect.right / 2, 0);

            pDC->SelectObject(DarkShadowPen);
            pDC->MoveTo(rect.right / 2 + 2, 3);
            pDC->LineTo(rect.right + 1, rect.bottom + 1);

            pDC->MoveTo(rect.right - 1, rect.bottom);
            pDC->LineTo(1, rect.bottom);
        }
        //elseif
        //elseif
        break;
    } //case

    case POINT_DOWN :
    case POINT_LEFT :
    }

    //绘制按钮标签
    ...

    pDC->RestoreDC(nSavedDC);
}

```

在 `DrawItem` 函数中首先得到按钮的位置和风格信息。然后,根据按钮原先的外形——矩形得到相应三角形的定点坐标。此后根据按钮不同方向(`POINT_UP`、`POINT_RIGHT`、`POINT_LEFT` 和 `POINT_DOWN`)以及不同风格,选择绘制三角形边线的不同顺序和画笔。这里就要解释一下为什么要区分三角形的方向了。读者可以从代码中看到,在 `DrawItem` 函数中得到按钮原来的边界矩形后,首先将其转化为正方形,然后根据这个正方形计算三角形的坐标。具体的计算方法为:选择其中一边的中点作为三角形的一个顶

点,并选择其对边的两个顶点作为三角形的另外两个顶点。而这个中点就是三角形的方向,也就是选择正方形的哪条边的中点的问题。

这里有个问题:虽然绘制出的按钮为三角形,但是其默认的形状为矩形。那么当用户单击默认矩形区域中(但绘制的三角形按钮外)时,也会导致按钮被触发。为了解决这个问题,在 CTriangleButton 类中重载了 PreSubclassWindow 函数。PreSubclassWindow 函数主要用于在窗口(按钮窗口)被归属于某类前,进行一些必要的动态修改。在本例中,就是重载该函数以改变按钮的默认区域。清单 2-9 为 PreSubclassWindow 函数的源代码:

清单 2-9 PreSubclassWindow() 函数

```
void CTriangleButton::PreSubclassWindow()
{
    CButton::PreSubclassWindow();

    CRect rect;
    GetClientRect(rect);
    rect.bottom = rect.right = min(rect.bottom, rect.right);
    rect.bottom -= rect.bottom % 2; rect.right -= rect.right % 2;

    SetWindowPos(NULL, 0, 0, rect.right, rect.bottom, SWP_NOMOVE | SWP_NOZORDER);

    CPoint Head, RightLeg, LeftLeg;

    switch (PointDirection) {
        case POINT_UP :
            Head.x = rect.right / 2; Head.y = 0;
            RightLeg.x = rect.right; RightLeg.y = rect.bottom;
            LeftLeg.x = 0; LeftLeg.y = rect.bottom;
            break;
        case POINT_DOWN :
            Head.x = rect.right / 2; Head.y = rect.bottom;
            RightLeg.x = 0; RightLeg.y = 0;
            LeftLeg.x = rect.right; LeftLeg.y = 0;
            break;
        case POINT_LEFT :
            Head.x = 0; Head.y = rect.bottom / 2;
            RightLeg.x = rect.right; RightLeg.y = 0;
            LeftLeg.x = rect.right; LeftLeg.y = rect.bottom;
            break;
        case POINT_RIGHT :
            Head.x = rect.right; Head.y = rect.bottom / 2;
            RightLeg.x = 0; RightLeg.y = rect.bottom;
            LeftLeg.x = 0; LeftLeg.y = 0;
            break;
        default :
            ASSERT(FALSE);
    } //switch

    CPoint points[3];
    points[0] = Head; points[1] = RightLeg; points[2] = LeftLeg;
```



```
SetWindowRgn(NULL, FALSE);  
  
CurrentRegion.DeleteObject();  
CurrentRegion.CreatePolygonRgn(points, 3, ALTERNATE);  
  
SetWindowRgn(CurrentRegion, TRUE);  
  
ModifyStyle(0, BS_OWNERDRAW);  
|
```

与 DrawItem 函数类似,首先得到按钮的边界矩形(这里使用了不同的方法: GetClientRect)。然后根据该矩形以及按钮的方向,确定按钮三角形的各个顶点坐标。再根据这些坐标创建一个区域,并调用 SetWindowRgn 函数将该区域(三角形)设置为按钮窗口,这样就解决了上述问题。

在实际使用 CTriangleButton 类时,只要在应用程序中将对应的按钮声明为 CTriangleButton 对象即可。配套光盘的 chap3/trianglebtn 目录为使用该类的示例工程。该示例允许动态改变按钮的文本和按钮方向。其运行结果如图 2-6 所示。

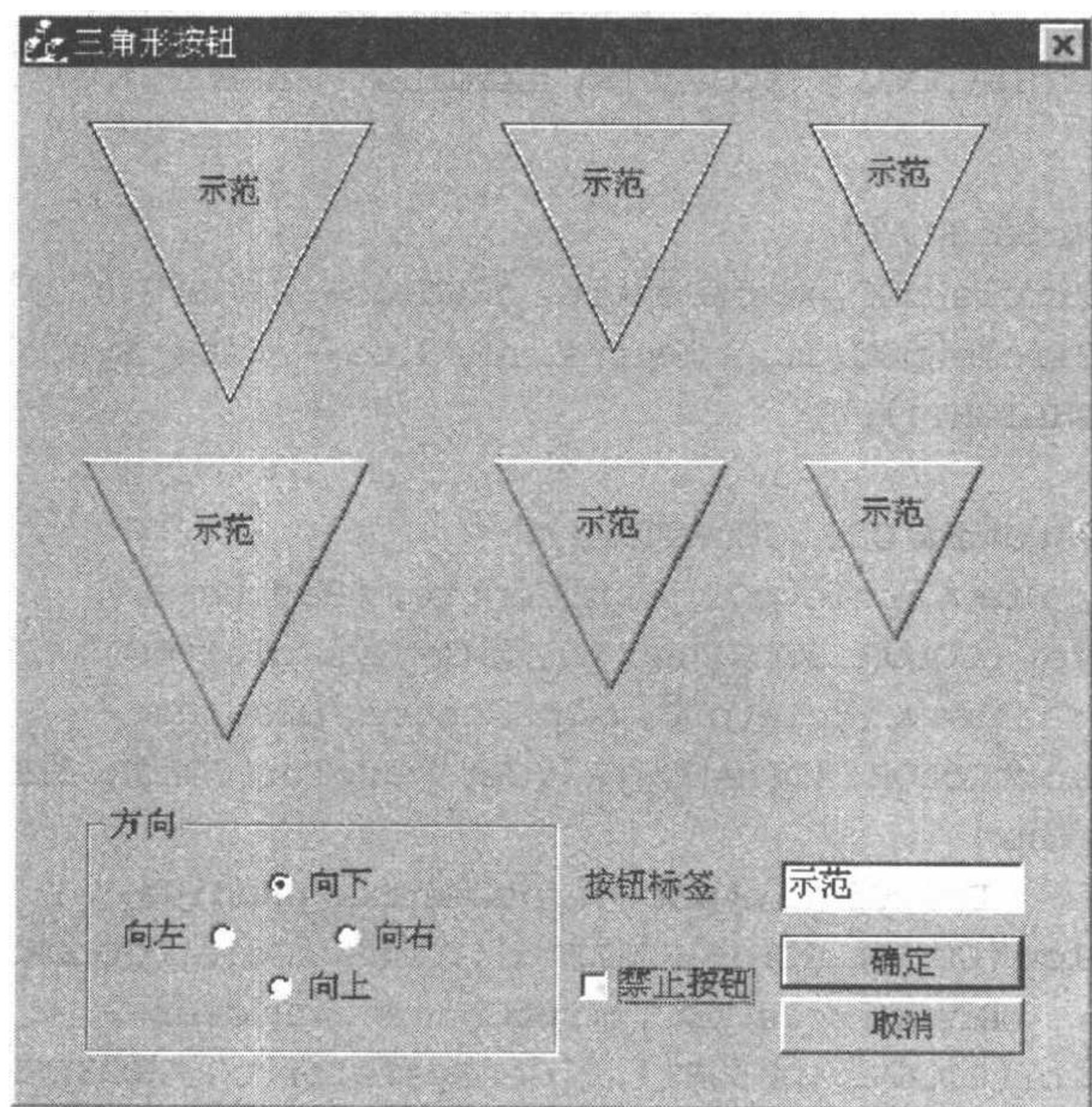


图 2-6 三角形按钮示例

2.3.2 创建圆形按钮

圆形是一种特殊的多边形,圆形按钮的创建方法也与多边形非常类似,并且相对来说要简单一些,因为它不涉及方向的问题。这里给出圆形按钮的例子,主要是为了与多边形和后面球形例子对比。首先设计一个圆形按钮管理类: CRoundButton。该类是 CButton 类的派生类,主要功能是在重载的 DrawItem 函数中完成的。CRoundButton 类的定义与 CTriangleButton 类的定义基本一致,我们需要关心的主要是其绘制方法,也就是 DrawItem 函数

的实现。清单 2-10 为 DrawItem 函数的源代码：

清单 2-10 DrawItem() 函数

```
void CRoundButton::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
    ASSERT(lpDrawItemStruct != NULL);

    CDC * pDC = CDC::FromHandle(lpDrawItemStruct->hDC);
    CRect rect = lpDrawItemStruct->rcItem;
    UINT state = lpDrawItemStruct->itemState;
    UINT nStyle = GetStyle();
    int nRadius = m_nRadius;

    int nSavedDC = pDC->SaveDC();

    pDC->SelectStockObject(NULL_BRUSH);
    pDC->FillSolidRect(rect, ::GetSysColor(COLOR_BTNFACE));

    // 如果按钮有焦点,则绘制焦点圆形
    if ((state & ODS_FOCUS) && m_bDrawDashedFocusCircle)
        DrawCircle(pDC, m_ptCentre, nRadius--, RGB(0,0,0));

    // 绘制按钮
    if (nStyle & BS_FLAT) {
        DrawCircle(pDC, m_ptCentre, nRadius--, RGB(0,0,0));
        DrawCircle(pDC, m_ptCentre, nRadius--, ::GetSysColor(COLOR_3DHIGHLIGHT));
    } else {
        if ((state & ODS_SELECTED)) {
            DrawCircle(pDC, m_ptCentre, nRadius--,
                ::GetSysColor(COLOR_3DDKSHADOW), ::GetSysColor(COLOR_3DHIGHLIGHT));
            DrawCircle(pDC, m_ptCentre, nRadius--,
                ::GetSysColor(COLOR_3DSHADOW), ::GetSysColor(COLOR_3DLIGHT));
        } else {
            DrawCircle(pDC, m_ptCentre, nRadius--,
                ::GetSysColor(COLOR_3DHIGHLIGHT), ::GetSysColor(COLOR_3DDKSHADOW));
            DrawCircle(pDC, m_ptCentre, nRadius--,
                ::GetSysColor(COLOR_3DLIGHT), ::GetSysColor(COLOR_3DSHADOW));
        }
    }

    // 绘制按钮文本
    CString strText;
    GetWindowText(strText);

    if (! strText.IsEmpty())
    {
        CRgn rgn;
        rgn.CreateEllipticRgn(m_ptCentre.x-nRadius, m_ptCentre.y-nRadius,
            m_ptCentre.x+nRadius, m_ptCentre.y+nRadius);
    }
}
```

```

pDC->SelectClipRgn(&rgn);

CSize Extent = pDC->GetTextExtent(strText);
CPoint pt = CPoint(m_ptCentre.x - Extent.cx/2, m_ptCentre.x -
Extent.cy/2 );

if (state & ODS_SELECTED) pt.Offset(1,1);

pDC->SetBkMode(TRANSPARENT);

if (state & ODS_DISABLED)
    pDC->DrawState(pt, Extent, strText, DSS_DISABLED, TRUE, 0,
    (HBRUSH)NULL);
else
    pDC->TextOut(pt.x, pt.y, strText);

pDC->SelectClipRgn(NULL);
rgn.DeleteObject();
}

if ((state & ODS_FOCUS) && m_bDrawDashedFocusCircle)
    DrawCircle(pDC, m_ptCentre, nRadius - 2, RGB(0,0,0), TRUE);

pDC->RestoreDC(nSavedDC);
}

```

配套光盘的 chap3/circlebtn 目录为使用 CRoundButton 类的示例工程,其运行结果如图 2-7 所示。

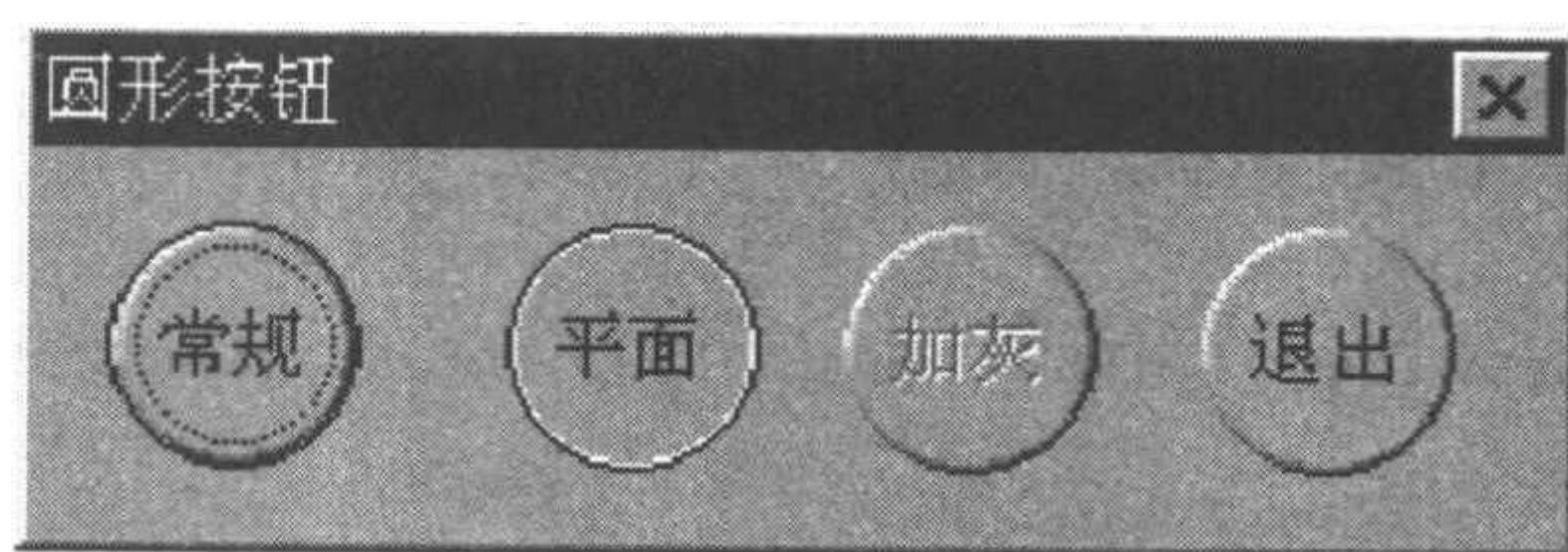


图 2-7 圆形按钮示例

2.3.3 创建球形按钮

读者在使用一些软件时,可能会看到如图 2-8 所示的选项。也许读者也考虑过如何实现这样的选项,不过 Visual C++ 中并没有直接提供类似的控件。因此,必须自己手动实现。在本节中就向读者介绍如何利用按钮实现此类选项。

首先从 CButton 类中派生一个 CSphereButton 类,用于管理球形按钮的创建。可以预料,球形按钮的绘制依然要通过重载 DrawItem 函数实现。这里需要提醒读者注意的是,文本所在区域实际是原来的按钮。当按钮状态变化时,文本本身可以变化,但是其下的按钮区域却不能变化,因此必须预先设置窗口

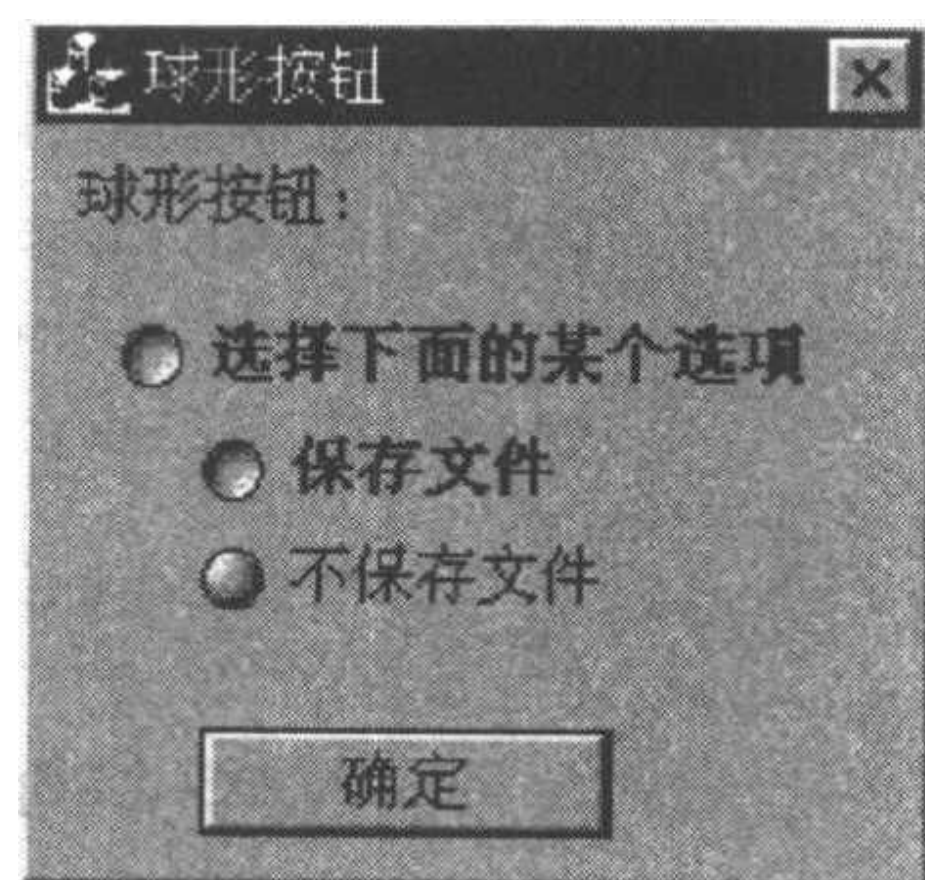


图 2-8 球形按钮

区域。同时,当单击文本所在位置处时,按钮不能作出响应;而只有当单击球形区域时,按钮和文本才能进行响应,这必须在对鼠标单击的消息响应函数中完成。

下面我们就一起完成 CSphereButton 类的设计,清单 2-11 所示为 CSphereButton 类的定义:

清单 2-11 CSphereButton 类定义

```
class CSphereButton : public CButton
{
// Construction
public:
    CCSphereButton( bool bDepressed = false, bool bCenterAlign = true );

// Attributes
public:
    bool MouseOverItem();

// Operations
public:
protected:
    COLORREF GetColor( double dAngle, COLORREF crBright, COLORREF crDark);
    void DrawCircle(CDC * pDC, CPoint p, LONG lRadius, COLORREF crColour, BOOL
                    bDashed = FALSE);
    void DrawCircle(CDC * pDC, CPoint p, LONG lRadius, COLORREF crBright,
                    COLORREF crDark);

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CBounceButton)
    public:
        virtual void DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct);
    protected:
        virtual void PreSubclassWindow();
    //}}AFX_VIRTUAL

// Implementation
public:
    int m_nMargin;
    int m_nRadius;

public:
    bool Depress( bool Down );
    bool IsDepressed();
    void Set_Check( int nCheck );
    virtual ~CBounceButton();

    // Generated message map functions
protected:
    //{{AFX_MSG(CSphereButton)
    afx_msg void OnKillFocus(CWnd * pNewWnd);
    afx_msg void OnLButtonDblClk(UINT nFlags, CPoint point);
    afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
```

```

afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
afx_msg void OnMouseMove(UINT nFlags, CPoint point);
//}}AFX_MSG
void OnMouseEnter(void);
void OnMouseLeave(void);

DECLARE_MESSAGE_MAP()

private:
    CRgn m_btnRgn;
    CPoint m_ptCenter;
    CRect m_textRect;
    CFont m_BoldFont;
    bool m_bMouseOver;
    bool m_bDepressed;
    bool m_bMouseTracking;
    bool m_bLMouseButtonDown;
    bool m_bDrawFocusRect;
    bool m_bCenterAlign;
};

```

类中定义了许多成员变量,它们主要用于标识鼠标状态和按钮状态,读者不必深究。只要理解了类的主要功能实现函数,它们的含义也就自然明白了。

前面已经说过,预设窗口区域的工作是在 PreSubclassWindow 函数中完成的,其源代码如清单 2-12 所示:

清单 2-12 PreSubclassWindow() 函数

```

void CSphereButton::PreSubclassWindow()
{
    CButton::PreSubclassWindow();

    // 将按钮类型修改为自绘制类型
    ModifyStyle(0, BS_OWNERDRAW);

    CRect rect;
    GetClientRect(rect);

    // 设置按钮尺寸
    LOGFONT lf;
    GetFont() -> GetLogFont(&lf);
    m_nRadius = lf.lfHeight;
    if( m_nRadius == 0 )
        m_nRadius = 15;
    if( m_nRadius < 0 )
        m_nRadius = (-1) * m_nRadius;
    m_nRadius = (int)(rect.bottom * 0.5) - 5;

    // 如果需要可以后改变 m_nRadius
    if( m_nRadius > 6 )
        m_nRadius = 6;
}

```



```

m_ptCenter.x = rect.left + m_nRadius + 1;
if( m_bCenterAlign )
{
    m_ptCenter.y = rect.top + (int)(rect.Height() * 0.5);
}
else
{
    m_ptCenter.y = rect.top + m_nRadius + 1;
}

m_btnRgn.CreateEllipticRgn(rect.left, m_ptCenter.y - m_nRadius - 1, rect.left
                           + (2 * m_nRadius) + 4,
                           m_ptCenter.y + m_nRadius + 3 );
// 得到当前窗口字体,并用以创建黑体字体
LOGFONT logfont;
CFont * pWndFont = GetFont();
pWndFont -> GetLogFont( &logfont );
logfont.lfWeight = 700; // set the weight to a bold value
m_BoldFont.CreateFontIndirect( &logfont );
}

```

在 `PreSubclassWindow` 函数中,首先调用 `GetClientRect` 函数得到按钮的原始尺寸,然后据此设置字体高度和球的半径。最后将球设置为按钮窗口区域。在以后的 `DrawItem` 函数中将使用这些值进行绘制。`DrawItem` 函数的部分源代码如清单 2-13 所示。

清单 2-13 `DrawItem()` 函数

```

void CSphereButton::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
    ASSERT(lpDrawItemStruct != NULL);

    CDC * pDC = CDC::FromHandle(lpDrawItemStruct -> hDC);
    CRect rect = lpDrawItemStruct -> rcItem;
    UINT state = lpDrawItemStruct -> itemState;
    UINT nStyle = GetStyle();
    int nRadius = m_nRadius;

    int nSavedDC = pDC -> SaveDC();

    pDC -> SelectStockObject(NULL_BRUSH);
    pDC -> FillSolidRect(rect, ::GetSysColor(COLOR_BTNFACE));

    // 如果按钮被禁止,则绘制其边界圆形
    if( lpDrawItemStruct -> itemState & ODS_DISABLED )
    {
        DrawCircle(pDC, m_ptCenter, nRadius + 1, ::GetSysColor(COLOR_BTNHIGHLIGHT));
        nRadius -= 4;
    }
    else
    {
        DrawCircle(pDC, m_ptCenter, nRadius + 1, ::GetSysColor(COLOR_WINDOW));
    }
}

```

```

DOWFRAME));

// 如果按钮不是平面风格,则绘制其边框
if (nStyle & BS_FLAT)
{
    DrawCircle(pDC, m_ptCenter, nRadius--, RGB(0,0,0));
    DrawCircle(pDC, m_ptCenter, nRadius--, ::GetSysColor(COLOR_3DHIGHLIGHT));
}
else
{
    if (m_bDepressed)    //如果按钮是按下状态
    {
        DrawCircle(pDC, m_ptCenter, nRadius--,
            ::GetSysColor(COLOR_3DDKSHADOW), ::GetSysColor(COLOR_3DHIGHLIGHT));
        DrawCircle(pDC, m_ptCenter, nRadius--,
            ::GetSysColor(COLOR_3DDKSHADOW), ::GetSysColor(COLOR_3DHIGHLIGHT));
        DrawCircle(pDC, m_ptCenter, nRadius--,
            ::GetSysColor(COLOR_3DSHADOW), ::GetSysColor(COLOR_3DLIGHT));
        DrawCircle(pDC, m_ptCenter, nRadius--,
            ::GetSysColor(COLOR_3DSHADOW), ::GetSysColor(COLOR_3DLIGHT));
        pDC->SelectObject(m_BoldFont);
    }
    else //如果按钮为弹起态
    {
    }
}

// 绘制文本
CString strText;
GetWindowText(strText);
CSize Extent;
CPoint pt;

CRect textRect, clientRect;
GetClientRect(clientRect);
textRect = clientRect;
textRect.left += (2 * (m_nRadius + 5));
textRect.right -= 2;
if (! strText.IsEmpty())
{
    Extent = pDC->GetTextExtent(strText);
    pt.x = rect.left + (2 * nRadius) + m_nMargin;
    pt.y = (int)((rect.Height() - Extent.cy) * 0.5);

    pDC->SetBkMode(TRANSPARENT);

    if (state & ODS_DISABLED)

```

```

        pDC->DrawState(pt, Extent, strText, DSS_DISABLED, TRUE, 0,
            (HBRUSH)NULL);
    else
    {
        pDC->DrawText( strText, textRect,
            DT_LEFT|DT_NOPREFIX|DT_WORDBREAK|DT_CALCRECT );
        int h = textRect.Height();
        textRect.top = clientRect.top + (int)((clientRect.Height()-
            textRect.Height())*0.5);
        textRect.bottom = textRect.top + h;
        pDC->DrawText( strText, textRect, DT_LEFT|DT_NOPREFIX|DT_WORD-
            BREAK );
        m_textRect = textRect;
    }
}
textRect.right += 2;
// 绘制控件的焦点矩形
POINT point;
GetCursorPos( &point );
if (m_bDrawFocusRect )
{
    textRect.InflateRect(1,1);
    textRect.right ++ ;
    pDC->DrawFocusRect( textRect );
    m_textRect = textRect;
}
pDC->RestoreDC(nSavedDC);
}

```

读者可以看到函数中使用了多个分支和条件语句,用于根据按钮的具体状态进行绘制。而按钮的状态标识则是通过消息处理函数设置。例如清单 2-14 为对鼠标左键单击的消息响应函数源代码:

清单 2-14 OnLButtonDown()函数

```

void CBounceButton::OnLButtonDown(UINT nFlags, CPoint point)
{
    // 确定鼠标在按钮上
    POINT pt;
    pt.x = point.x;
    pt.y = point.y;

    if( m_bDrawFocusRect )
    {
        OnLButtonDown = true;
        m_bDepressed = ! m_bDepressed;
        SetFocus();
        Invalidate();

        CButton::OnLButtonDown(nFlags, point);
    }
}

```

在其中,将鼠标被按下的标志 `OnLButtonDown` 设置为真;将按钮是否按下的标志 `m_bDepressed`取反;并设置按钮的焦点状态。最后调用 `Invalidate` 通知按钮控件重新绘制,这时框架将调用 `DrawItem` 函数,并使用这些状态标志参数绘制按钮。其他消息处理函数的原理也类似。这里需要向读者强调的是,在定制自绘制控件时需要考虑全面。因为,此时按钮的所有绘制工作都必须通过定制代码完成,因此也必须完整考虑所有情况。在实际开发中,需要反复测试,以免出错。配套光盘的 `chap3 \ spherebtn` 目录为球形按钮的源代码,读者请自行参阅。

2.4 动态创建高级按钮

前两节主要介绍了如何改变按钮颜色和形状,本节将在这个基础上创建一个高级按钮管理类。该类能够使用复杂的形状动态创建按钮,并且支持按钮的“滞留”选择等功能。

2.4.1 设计高级按钮管理类

在3.3节中介绍了创建多边形或圆形按钮,这些按钮形状比较简单。那么能不能创建复杂形状的按钮呢?例如,使用箭头形状作为按钮。可以想像如果使用创建多边形按钮的方法,将会非常复杂。这时,我们可以使用路径 `CRgn` 这个非常有用,但是经常被忽略的工具。另外,读者也许记得前面的例子中的类只能管理静态按钮,而不支持 `Create` 函数的动态创建。本节将向读者介绍如何支持动态创建。清单2-15所示为 `CAdvButton` 类的定义:

清单 2-15 `CAdvButton` 类的定义

```
class CAdvButton : public CButton
{
public:
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CAdvButton)
    public:
        virtual void DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct);
    protected:
        virtual void PreSubclassWindow();
        virtual LRESULT DefWindowProc(UINT message, WPARAM wParam, LPARAM lParam);
    //}}AFX_VIRTUAL

protected:
    //{{AFX_MSG(CAdvButton)
    afx_msg BOOL OnEraseBkgnd(CDC * pDC);
    afx_msg void OnMouseMove(UINT nFlags, CPoint point);
```



```

afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
//{{AFX_MSG
DECLARE_MESSAGE_MAP()
private:
    UINT m_nBorder;    //高亮显示时的边界宽度
    LONG m_lfEscapement;    // 按钮标签的方向
    COLORREF m_nColor, m_sColor, m_hColor, m_dColor; //按钮不同状态下的背景色
    CBitmap * m_pNormal;    //按钮所用的位图
    CBitmap * m_pSelected;
    CBitmap * m_pHover;
    CBitmap * m_pDisabled;
    CPoint m_CenterPoint;    //按钮标签的中点
    BOOL m_bMouseDown;    //标识按下鼠标
    BOOL m_bHover;    //标识鼠标在按钮上
    BOOL m_bCapture;    //标识鼠标已经捕捉到按钮
    HRGN m_hRgn;    //指定区域
    BOOL m_bNeedBitmaps;    //标识是否需要重构状态位图
    void DrawButton(CDC * pDC, CRect * pRect, UINT state);    //绘制按钮
    void PrepareStateBitmaps(CDC * pDC, CRect * pRect);    //准备按钮位图
    BOOL HitTest(CPoint point);    //检查某点是否在按钮区域中
    void RgnPixelWork(CDC * pDC, CRgn * pRgn);
    void FrameRgn3D(HDC hDC, const HRGN hRgn, BOOL bSunken);    //将显示 3d 阴影的区域
    void CheckHover(CPoint point);
protected:
    void PrepareNormalState(CDC * pDC, CDC * pMemDC, CRect * pRect); //准备状态位图
    void PrepareSelectedState(CDC * pDC, CDC * pMemDC, CRect * pRect);
    void PrepareHoverState(CDC * pDC, CDC * pMemDC, CRect * pRect);
    void PrepareDisabledState(CDC * pDC, CDC * pMemDC, CRect * pRect);
    void DrawButtonCaption(HDC hDC, CRect * pRect, BOOL bEnabled, BOOL bSunken);
    void PaintRgn(CDC * pDC, CDC * pMemDC, CBitmap * pBitmap, COLORREF color,
                  CRect * pRect, BOOL bEnabled, BOOL bSunken);
public:
    CAdvButton();
    virtual ~CAdvButton(); // destructor
    BOOL Create(LPCTSTR lpszCaption, DWORD dwStyle, const CPoint point, const
               HRGN hRgn,
               CWnd * pParentWnd, UINT nID);    //创建具有默认边界和颜色的按钮
    BOOL Create(LPCTSTR lpszCaption, DWORD dwStyle, const CPoint point, const
               HRGN hRgn,
               CWnd * pParentWnd, UINT nID, COLORREF color);    //创建具有“停滞”选择和选中
                                                                颜色的按钮
    BOOL Create(LPCTSTR lpszCaption, DWORD dwStyle, const CPoint point, const
               HRGN hRgn,

```

```

        CWnd * pParentWnd, UINT nID, UINT nBorder, COLORREF nColor,
        COLORREF
        sColor, COLORREF hColor, COLORREF dColor);    //创建具有多种风格
                                                    的按钮
    BOOL Create(LPCTSTR lpszCaption, DWORD dwStyle, const CPoint point, const
    HRGN hRgn,
    CWnd * pParentWnd, UINT nID, UINT nBorder, LONG lfEscapement, COLORREF
    nColor,
    COLORREF sColor, COLORREF hColor, COLORREF dColor);    //上面函数的另一种方式
};

```

读者可以看到,这个类比较复杂,但同时也提供了更为强大的功能。下面的内容将对该类详细说明。

2.4.2 动态创建

CAdvButton 类中提供了 4 个版本的 Create 函数,它们分别用于创建不同风格和颜色的按钮。这里介绍其中最底层的版本,其他版本都需要调用这个版本(实际上其他版本的主要功能是初始化附加成员,例如按钮“停滞”选择颜色)。清单 2-16 所示为 Create 函数的源代码:

清单 2-16 Create()函数

```

BOOL CAdvButton::Create(LPCTSTR lpszCaption, DWORD dwStyle, const CPoint point,
const HRGN
hRgn, CWnd * pParentWnd, UINT nID)
{
    // 使用区域参数初始化成员变量
    DeleteObject(m_hRgn);
    m_hRgn = CreateRectRgn(0, 0, 31, 31);
    CRect box(0, 0, 0, 0);
    if (m_hRgn != 0)
        CombineRgn(m_hRgn, hRgn, 0, RGN_COPY);

    // 确保区域的边界矩形左上角坐标为 (0, 0)
    GetRgnBox(m_hRgn, &box);
    OffsetRgn(m_hRgn, -box.left, -box.top);
    GetRgnBox(m_hRgn, &box);

    // 设置按钮标签输出位置
    m_CenterPoint = CPoint(box.left + box.Width() / 2, box.top + box.Height()
/ 2);
    box.OffsetRect(point);

    return CButton::Create(lpszCaption, dwStyle, box, pParentWnd, nID);
}

```

其中, lpszCaption 参数指定了按钮文本; dwStyle 参数指定了标准按钮风格; point 参数指定了按钮控件在父窗口中的位置; hRgn 参数指定了按钮的外形; pParentWnd 参数指定

了父窗口句柄; `nID` 指定了按钮参数的标识符。

在函数中,首先使用区域参数初始化类本身的区域成员,以用于此后的按钮绘制。然后修改路径的边界矩形坐标,使其左上角坐标为(0,0)。改变坐标的主要原因是: `hRgn` 参数为父窗口坐标,而在 `CAdvButton` 类中 `hRgn` 的边界矩形应该为整个按钮的边界矩形,其左上角当然应该为原点。此后,在设置了文本输出位置后,调用 `CButton::Create` 函数完成按钮控件的创建。

此时,读者应该明白调用 `Create` 函数完毕后,实际创建的依然是 Windows 标准按钮控件,其位置和尺寸由区域的边界矩形给定。因此,定制的按钮形状的显示依然需要通过重载 `DrawItem` 函数完成。

由于必须自绘制按钮控件,因此要保证按钮为 Owner-Draw 风格,这个工作是通过在 `PreSubClassWindow` 函数中,调用 `ModifyStyle` 函数完成的,其源代码如清单 2-17 所示:

清单 2-17 `PreSubClassWindow()` 函数

```
void CAdvButton::PreSubclassWindow()
{
    ModifyStyle(0, BS_OWNERDRAW | BS_PUSHBUTTON);
    CButton::PreSubclassWindow();
}
```

此外,在创建按钮窗口时,要将其区域改变为调用 `Create` 函数时指定的区域,这可以通过 `SetWindowRgn` 函数完成,其源代码如清单 2-18 所示:

清单 2-18 `OnCreate()` 函数

```
int CAdvButton::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CButton::OnCreate(lpCreateStruct) == -1)
        return -1;

    m_bNeedBitmaps = true;
    SetWindowRgn(m_hRgn, true);

    return 0;
}
```

2.4.3 按钮绘制过程分析

与其他自绘制按钮的管理类一样,掌握其功能和设计思想也要从 `DrawItem` 入手。下面我们将按照按钮的绘制顺序向读者介绍此类。清单 2-19 所示为 `DrawItem` 函数的源代码:

清单 2-19 `DrawItem()` 函数

```
void CAdvButton::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
    CDC * pDC = CDC::FromHandle(lpDrawItemStruct->hDC);
    CRect rect;
```

```

    GetClientRect(rect);

    if (m_bNeedBitmaps)
        PrepareStateBitmaps(pDC, &rect);

    DrawButton(pDC, &rect, lpDrawItemStruct->itemState);
}

```

该类中的 DrawItem 函数的形式比较简单,它的主要绘制动作都被包装到单独的函数中了。如果按钮需要状态位图,则调用 PrepareStateBitmaps 准备状态位图,然后调用 DrawButton 函数绘制按钮。清单 2-20 所示为 PrepareStateBitmaps 函数的源代码:

清单 2-20 PrepareStateBitmaps() 函数

```

void CAdvButton::PrepareStateBitmaps(CDC * pDC, CRect * pRect)
{
    CDC * pMemDC;
    pMemDC = new CDC;
    pMemDC->CreateCompatibleDC(pDC);

    PrepareNormalState(pDC, pMemDC, pRect);
    PrepareSelectedState(pDC, pMemDC, pRect);
    PrepareHoverState(pDC, pMemDC, pRect);
    PrepareDisabledState(pDC, pMemDC, pRect);

    delete pMemDC;
    m_bNeedBitmaps = false;
}

```

函数中分别调用 PrepareNormalState、PrepareSelectedState、PrepareHoverState 和 PrepareDisabledState 函数准备按钮不同状态所需的位图。读者也许会问,在函数开始时创建了临时设备环境 pMemDC,在函数结尾就被删除了,那么这个函数有意义吗?实际上绘制动作分别在上述 4 个函数中完成。而在函数结尾时,绘制已经完成,因此将其删除。

这里以 PrepareDisabledState(其他类似)为例,其源代码如清单 2-21 所示:

清单 2-21 PrepareDisabledState() 函数

```

void CAdvButton::PrepareDisabledState(CDC * pDC, CDC * pMemDC, CRect * pRect)
{
    delete m_pDisabled;
    m_pDisabled = new CBitmap;
    PaintRgn(pDC, pMemDC, m_pDisabled, m_dColor, pRect, false, false);
}

```

其中 PaintRgn 函数负责由位图生成路径,其源代码如清单 2-22 所示:

清单 2-22 PaintRgn() 函数

```

void CAdvButton::PaintRgn(CDC * pDC, CDC * pMemDC, CBitmap * pBitmap, COLORREF color,
    CRect * pRect, BOOL bEnabled, BOOL bSunken)
{
    // 创建位图

```



```

pBitmap -> CreateCompatibleBitmap(pDC, pRect -> Width(), pRect -> Height
());
CBitmap * pOldBitmap = pMemDC -> SelectObject(pBitmap);

// 准备区域
HRGN hRgn = CreateRectRgn(0, 0, 0, 0);
GetWindowRgn(hRgn);

// 使用透明色填充矩形和区域
HBRUSH hBrush = CreateSolidBrush(color);
pMemDC -> FillSolidRect(pRect, RGB(0, 0, 0));
FillRgn(pMemDC -> GetSafeHdc(), hRgn, hBrush);
DeleteObject(hBrush);

// 绘制三维边界和按钮标签
DrawButtonCaption(pMemDC -> GetSafeHdc(), pRect, bEnabled, bSunken);
FrameRgn3D(pMemDC -> GetSafeHdc(), hRgn, bSunken);

// 清除区域
DeleteObject(hRgn);
pMemDC -> SelectObject(pOldBitmap);
}

```

在 `PaintRgn` 函数中,将按钮窗口区域(路径)绘制到相应位图中。这样在以后绘制按钮时,只需根据不同状态将对应位图选入按钮设备环境中即可。在其中调用 `DrawButtonCaption` 函数将按钮文本绘制在位图中;而调用 `FrameRgn3D` 函数绘制按钮的边框。实际上,绘制到不同状态位图上的图形和文本都是一样的,只是其使用的颜色和边框不同,例如禁止状态使用灰色。这两个函数比较简单,只是完成单一的绘制工作,请读者参考配套光盘 `chap3\advancebtn` 下的源代码。

`DrawButton` 函数负责根据按钮状态,将不同的位图选入设备环境,清单 2-23 所示为函数的源代码:

清单 2-23 DrawButton() 函数

```

void CAdvButton::DrawButton(CDC * pDC, CRect * pRect, UINT state)
{
    // 创建内存设备环境
    CDC * pMemDC = new CDC;
    pMemDC -> CreateCompatibleDC(pDC);
    CBitmap * pOldBitmap;

    // 得到窗口区域
    HRGN hRgn = CreateRectRgn(0, 0, 0, 0);
    GetWindowRgn(hRgn);

    // 根据按钮状态选择位图
    if (state & ODS_DISABLED)
        pOldBitmap = pMemDC -> SelectObject(m_pDisabled);
    else {
        if (state & ODS_SELECTED)
            pOldBitmap = pMemDC -> SelectObject(m_pSelected);
    }
}

```

```

        else {
            if (m_bHover)
                pOldBitmap = pMemDC -> SelectObject(m_pHover);
            else
                pOldBitmap = pMemDC -> SelectObject(m_pNormal);
        }
    }

    // 使用区域进行裁剪
    ::SelectClipRgn(pDC -> GetSafeHdc(), hRgn);
    pDC -> BitBlt(0, 0, pRect -> Width(), pRect -> Height(), pMemDC, 0, 0, SRC-
    COPY);
    ::SelectClipRgn(pDC -> GetSafeHdc(), NULL);

    DeleteObject(hRgn);
    pMemDC -> SelectObject(pOldBitmap);
    delete pMemDC;
}

```

需要指出的是,先将图形绘制在后备位图上,然后在需要时将其拷贝到当前设备环境中,这是一种非常通用的手段,在许多绘图软件中也会采用。

2.4.4 使用高级按钮管理类

使用 CAdvButton 类的方法非常简单,只要在应用程序中(例如对话框应用程序中)声明 CAdvButton 对象,然后创建合适的区域对象(亦即按钮外形),并传递给 CAdvButton 对象的 Create 函数(通常在对话框的 OnInitDialog 函数中进行),示范代码如清单 2-24 所示:

清单 2-24 OnInitDialog() 函数

```

BOOL CAdvButtonDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // 矩形按钮
    r = CreateRectRgn(0, 0, 63, 31);
    m_Btn1.Create("按钮 1", WS_CHILD | WS_VISIBLE, CPoint(15, 15), r, this, MY_
    BTN1, RGB(255, 255, 0));
    DeleteObject(r);

    // 椭圆按钮
    r = CreateEllipticRgn(0, 0, 63, 31);
    m_Btn2.Create("按钮 2", WS_CHILD | WS_VISIBLE, CPoint(95, 15), r, this, MY_
    BTN2, 2,
    GetSysColor(COLOR_BTNFACE), RGB(0, 255, 0), RGB(0, 255, 0),
    GetSysColor(COLOR_BTNFACE));
    DeleteObject(r);

    // 创建被拉伸的椭圆按钮
    HRGN rgnR = CreateRectRgn(0, 0, 127, 31);
}

```

```

HRGN rgnE = CreateEllipticRgn(0, 0, 127, 31);
OffsetRgn(rgnR, 63, 0);
CombineRgn(rgnE, rgnE, rgnR, RGN_DIFF);
m_Btn3.Create("按钮 3", WS_CHILD | WS_VISIBLE, CPoint(175, 15), rgnE, this,
MY_BTN3, 2,
GetSysColor(COLOR_BTNFACE), RGB(156, 175, 194), RGB(237, 175, 71),
GetSysColor(COLOR_BTNFACE));
...
DeleteObject(rgnE);
DeleteObject(rgnR);

// 使用文本($ )创建按钮
HDC hDC = CreateCompatibleDC(GetDC() -> GetSafeHdc());
HRGN c;
LOGFONT lf;
GetFont() -> GetLogFont(&lf);
lf.lfHeight = -128;
lf.lfWeight = 1000;
HFONT hFont = CreateFontIndirect(&lf);
HFONT hOldFont = (HFONT) SelectObject(hDC, hFont);
c = CreateRectRgn(0, 0, 0, 0);

int mode = SetBkMode(hDC, TRANSPARENT);
BeginPath(hDC);
TextOut(hDC, 0, 0, "$", 1);
EndPath(hDC);
c = PathToRegion(hDC);
SetBkMode(hDC, mode);

m_Btn5.Create("", WS_CHILD | WS_VISIBLE, CPoint(15, 63), c, this, MY_BTN5,
RGB(255, 255, 0));
...
SelectObject(hDC, hOldFont);
DeleteObject(hFont);
DeleteObject(c);

// 创建多边形按钮
HRGN r1;
HRGN r2;
BeginPath(hDC);
MoveToEx(hDC, 0, 32, NULL);
LineTo(hDC, 48, 32);
LineTo(hDC, 48, 16);
LineTo(hDC, 96, 48);
LineTo(hDC, 48, 80);
LineTo(hDC, 48, 64);
LineTo(hDC, 0, 64);
LineTo(hDC, 0, 32);
EndPath(hDC);
cr = CreateRectRgn(0, 0, 63, 63);
cr = PathToRegion(hDC);

```

```

m_Btn9.Create("按钮", WS_CHILD | WS_VISIBLE, CPoint(32 + 64 + 96, 63 + 16
+ 128), cr, this,
MY_BTN9, 2, RGB(250, 207, 194), RGB(255, 0, 0), RGB(255, 0, 0),
GetSysColor(COLOR_BTNFACE));

DeleteObject(cr);
DeleteObject(hDC);

// 创建复杂区域
hDC = CreateCompatibleDC(GetDC() -> GetSafeHdc());
BeginPath(hDC);
MoveToEx(hDC, 31, 15, NULL);
CPoint p[7];
p[0] = CPoint(5, 0);
p[1] = CPoint(0, 55);
p[2] = CPoint(0, 28);
p[3] = CPoint(31, 64);
p[4] = CPoint(59, 55);
p[5] = CPoint(59, 0);
p[6] = CPoint(31, 15);

PolyBezier(hDC, &p[0], 7);
EndPath(hDC);
cr = CreateRectRgn(0, 0, 63, 63);
cr = PathToRegion(hDC);
m_Btn10.Create("按钮 10", WS_CHILD | WS_VISIBLE, CPoint(15, 63 + 16 + 64 +
64), cr, this,
MY_BTN10, 2, RGB(151, 244, 219), RGB(211, 247, 254), RGB(211, 247, 254),
GetSysColor(COLOR_BTNFACE));

DeleteObject(cr);
DeleteObject(hDC);

// 创建拉伸按钮
c = CreateRectRgn(16, 0, 80, 31);
r1 = CreateEllipticRgn(0, 0, 32, 32);
r2 = CreateEllipticRgn(64, 0, 96, 32);
CombineRgn(c, c, r1, RGN_OR);
CombineRgn(c, c, r2, RGN_OR);
m_Btn11.Create("按钮 11", WS_CHILD | WS_VISIBLE, CPoint(15 + 64, 63 + 32 +
64 + 64), c,
this, MY_BTN11, 2, GetSysColor(COLOR_BTNFACE), RGB(211, 247, 254), RGB
(211, 247, 254), GetSysColor(COLOR_BTNFACE));

DeleteObject(c);
DeleteObject(r1);
DeleteObject(r2);
...

return TRUE; // return TRUE unless you set the focus to a control
}

```

从上面的示范代码可以看出,使用区域对象能够非常容易地创建出复杂的按钮形状。

这与 CRng 类的强大功能是分不开的。例如,在创建字符按钮时,只要使用 BeginPath 和 EndPath 函数记录字符输出的路径,然后调用 PathToRegion 函数将字符转化为区域,此后使用该区域就能够创建出字符按钮。详细的源代码参见配套光盘 chap3 \ advbutton 目录下的工程文件,其执行结果如图 2-9 所示。

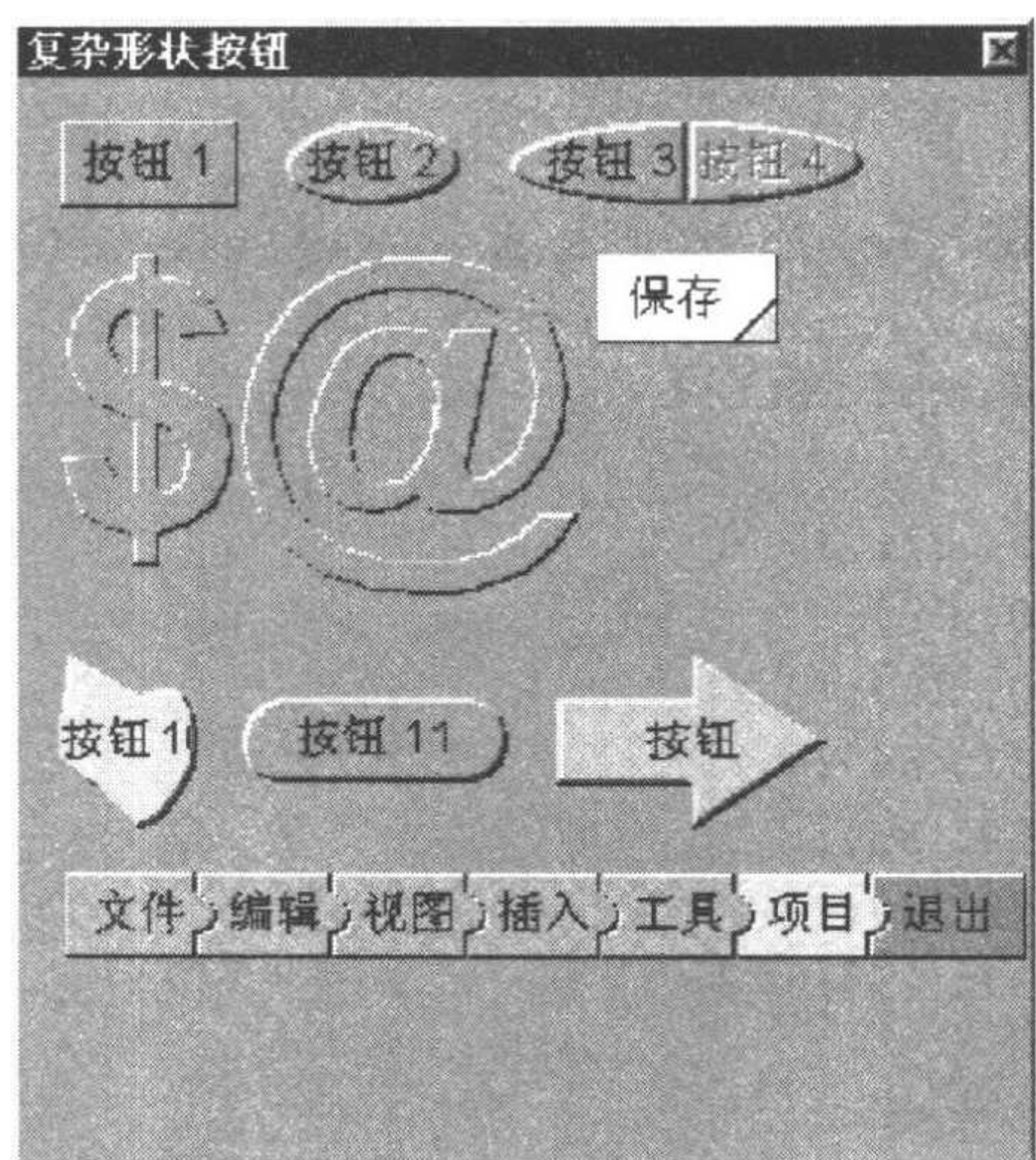


图 2-9 复杂形状按钮

本章小结

本章主要向读者介绍了如何修改常规 Windows 按钮控件,通过本章学习读者应该达到以下几点:

- 掌握 CButton 类的使用。
- 掌握定制按钮显示的方法。

第3章 编辑控件

编辑控件可以分为两种：CEdit 和 CRichEditCtrl,后者在文本编辑方面具有更强的功能。从用户界面的角度考虑,这两种控件的区别并不大,因此本章主要针对 CEdit 控件向读者介绍如何定制不同外观的编辑控件。

本章要点:

- CEdit 和 CEditView 类的使用;
- 定制不同类型的编辑控件。

3.1 编辑控件编程基础

编辑控件是一个用户可以在其中输入文本的矩形子窗口。CEdit 类为 Windows 编辑控件提供了功能支持,其派生结构如图 3-1 所示。

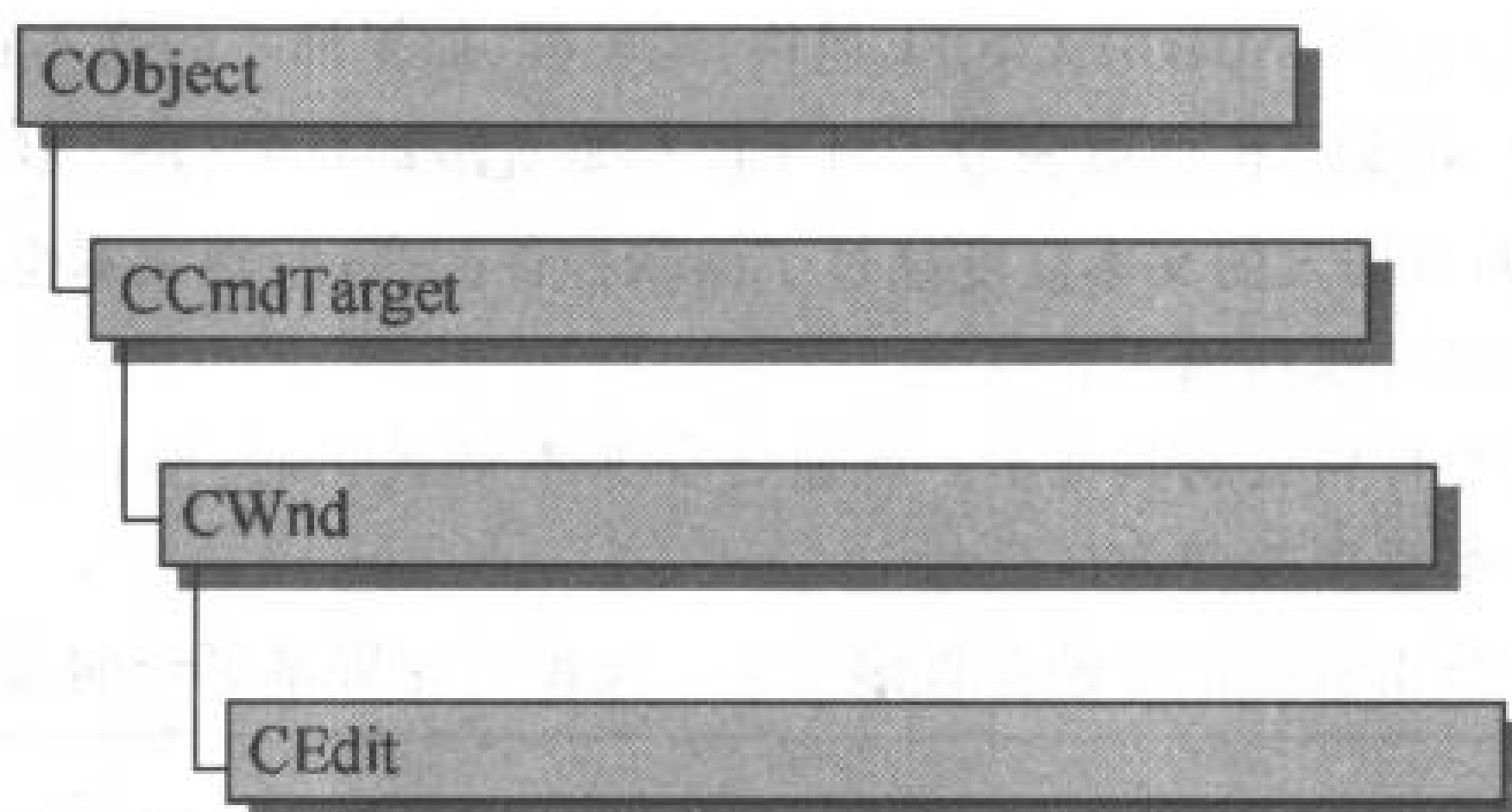


图 3-1 CEdit 类的派生结构

3.1.1 编辑控件概述

用户可以在对话框模板中创建编辑控件,或者直接在代码中创建。在两种情况下,首先都需要调用构造函数 CEdit 以创建 CEdit 对象,然后调用 Create 成员函数以创建 Windows 编辑控件,并将其与 CEdit 对象相联系。对于 CEdit 类的派生类来说,上述两个步骤可以简化为一步:在派生类中的构造函数中直接调用 Create 函数。

CEdit 继承了 CWnd 类的大多数功能。例如,调用 CWnd 类的 SetWindowText 和 GetWindowText 成员函数,可以设置整个编辑控件(即使是多行编辑控件)中的文本。如果要设置多行编辑控件中的部分文本,就需要调用 CEdit 类本身的成员函数: GetLine、SetSel、GetSel 和 ReplaceSel 等。

如果希望处理由按钮控件向其父窗口(通常对话框)发送的 Windows 通告消息,则需要相应的父窗口类中添加消息映射入口和消息处理函数。

每个消息映射入口都具有以下形式:

```
ON_Notification(id, memberFxn)
```

其中 id 指定了发送通告消息的控件 ID,而 memberFxn 则指定了用于处理控件通告消息的成员函数。

消息处理函数如下:

```
afx_msg void memberFxn( );
```

编辑控件常用的通告消息如表 3-1 所示。

表 3-1 编辑控件通告消息

通告	含义
EN_CHANGE	用户改变了编辑控件中的文本。与 EN_UPDATE 通告消息不同,该通告消息在 Windows 改变显示之前发送
EN_ERRSPACE	编辑控件无法分配所需内存
EN_HSCROLL	用户单击编辑控件的水平滚动条。父窗口在屏幕更新前就会接收到该通告消息
EN_KILLFOCUS	编辑控件失去了输入焦点
EN_MAXTEXT	当前插入的文本超出编辑控件允许的最大长度,并因此被截短。对于没有水平滚动条的(ES_AUTOHSCROLL 风格)编辑框,如果插入的文本长度超过了编辑控件的宽度,则文本也会被截短。对于没有垂直滚动条的(ES_AUTOVSCROLL 风格)编辑框,如果插入的文本高度超过了编辑控件的高度,则文本也会被截短
EN_SETFOCUS	编辑控件收到输入焦点
EN_UPDATE	编辑控件将更新其显示。该通告在文本格式化后,但在显示更新前发送,这样可以根据需要改变窗口尺寸
EN_VSCROLL	用户单击编辑控件的垂直滚动条。父窗口在屏幕更新前就会接收到该通告消息

如果在对话框中创建编辑控件,则编辑控件对象将在对话框关闭时自动销毁。如果在其他的 Windows 对象中使用嵌入编辑控件对象,也无需手动销毁它。只有当使用 new 函数在堆上创建编辑控件对象时,才需要在使用完毕后,调用 delete 函数销毁它。

3.1.2 构造函数

CEdit 类的构造函数包括: CEdit 和 Create,它们能够完成 CEdit 对象的构造和 Windows 编辑控件的创建等操作。

- CEdit
- 调用该函数以构造 CEdit 控件对象,其原型为:

```
CEdit();
```

- Create

调用该函数以创建 Windows 编辑控件,并将其与 CEdit 对象相联系,其原型为:

```
BOOL Create( DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID );
```

返回值:

如果函数调用成功,返回非零值,否则返回零值。

参数:

dwStyle —— 指定了编辑控件的风格,其取值可以为表 3-2 中列出各值的联合。

表 3-2 dwStyle 取值

dwStyle 取值	含义
ES_AUTOHSCROLL	当用户输入一个字符时,自动将文本向右滚动 10 个字符。当用户按下 Enter 键时,控件将滚动回位置 0
ES_AUTOVSCROLL	当用户按下 Enter 键时,编辑控件自动滚动到下一页
ES_CENTER	在单行或多行编辑控件中使文本居中对齐
ES_LEFT	在单行或多行编辑控件中使文本居左对齐
ES_LOWERCASE	将输入编辑控件中的所有字符转换为其大写形式
ES_MULTILINE	创建多行编辑控件(默认情况下,将创建单行编辑控件)。如果同时指定了 ES_AUTOVSCROLL 风格,则编辑控件将在用户按下 Enter 键时垂直滚动。如果没有指定 ES_AUTOVSCROLL 风格,则编辑控件将在按下 Enter 键时发出提示声音。如果指定了 ES_AUTOHSCROLL 风格,则在边界控件将在输入位置超过其右边界时自动水平滚动。
ES_NOHIDESEL	通常,编辑控件在失去输入焦点后,会隐藏其中被选文本的选中态;而当其再次获得输入焦点后,会恢复其中被选文本的选中态。指定了 ES_NOHIDESEL 风格将使编辑控件在失去输入焦点的时候,依然保持其中被选文本的选中态
ES_OEMCONVERT	指定该风格后,输入编辑框中的文本将被由 ANSI 字符集转换为 OEM 字符集,然后再将其转换为 ANSI 字符集。这确保当应用程序调用 AnsiToOem 函数时,编辑控件中的 ANSI 字符串能被正确地转换为 OEM 字符串。这种风格对于包含文件名的编辑控件尤其有用
ES_PASSWORD	指定该风格后,输入编辑控件中的字符都以 * 号显示。应用程序可以使用 SetPasswordChar 成员函数来改变显示的字符
ES_RIGHT	指定了编辑控件中的文本为右对齐风格
ES_UPPERCASE	将编辑控件中的所有字符都转换为其大写形式
ES_READONLY	指定该风格,将禁止用户对编辑框进行输入
ES_WANTRETURN	指定该风格后,当用户按下 Enter 键时,对话框中的多行编辑控件将插入“回车符”。如果不指定该风格,则当用户按下 Enter 键等价于按下对话框中的默认按钮。该风格对单行编辑控件无效

rect —— 指定了编辑控件的尺寸和位置,其类型可以为 CRect 对象或 RECT 结构。

pParentWnd —— 指定了编辑控件的父窗口,通常为对话框,该参数不能为 NULL。

nID —— 指定了编辑控件的 ID。

创建编辑控件需要两步:首先调用构造函数 CEdit,然后调用 Create 函数以创建

Windows编辑控件并将其与 CEdit 对象相连接。

当执行 Create 函数时,Windows 向编辑控件发送 WM_NCCREATE、WM_NCCALCSIZE、WM_CREATE 和 WM_GETMINMAXINFO 消息。默认情况下,这些消息由 CWnd 基类中的 OnNcCreate、OnNcCalcSize、OnCreate 和 OnGetMinMaxInfo 成员函数进行处理。如果需要扩展这些函数的功能,则需要一个 CEdit 的派生类,并在该类中重载这些消息的处理函数。

编辑控件能使用的窗口风格常数如表 3-3 所示。

表 3-3 编辑控件能使用的窗口风格常数

风格常数	含义
WS_CHILD	子窗口
WS_VISIBLE	窗口控件
WS_DISABLED	窗口被禁止
WS_GROUP	成组按钮
WS_TABSTOP	使按钮能够用 Tab 键切换

3.1.3 属性操作函数

CEdit 类的属性操作函数包括: CanUndo、GetLineCount、GetModify、SetModify、GetRect、GetSel、GetHandle、SetHandle、SetMargins、GetMargins、SetLimitText、GetLimitText、PosFromChar、CharFromPos、GetLine、GetPasswordChar 和 GetFirstVisibleLine,它们能够完成对 CEdit 对象属性的设置和查询等操作。

• CanUndo

调用该函数以确定编辑控件是否能够进行恢复操作,其原型为:

```
BOOL CanUndo( ) const;
```

返回值:

如果编辑控件中的最近一次操作,可以通过调用 Undo 函数恢复,则返回非零值,否则返回零值。

• GetLineCount

调用该函数以得到多行编辑控件中的文本的行数,其原型为:

```
int GetLineCount( ) const;
```

返回值:

如果多行编辑控件中没有内容,则返回值为 1。而如果有内容,则返回其行数。

• GetModify

调用该函数以确定编辑控件中的内容是否已经被修改,其原型为:

```
BOOL GetModify( ) const;
```

返回值:

如果编辑控件中的内容被修改,则返回非零值,否则返回零值。

Windows 自己负责维护一个用于标识编辑控件中内容是否被修改的内部标志。当编辑控件刚被创建时,该标志被清除以表示控件内容未被修改,调用 SetModify 函数也可以达到同样的目的。

- SetModify

调用该函数以设置或清除编辑控件的修改标志,其原型为:

```
void SetModify( BOOL bModified = TRUE );
```

参数:

bModified —— 如果该参数为 TRUE,则表示控件中的内容被修改;如果为 FALSE 则表示控件中的内容未改变。默认情况下,该标志为 FALSE。

- GetRect

调用该函数以得到编辑控件的格式矩形,其原型为:

```
void GetRect( LPRECT lpRect ) const;
```

参数:

lpRect —— 该参数将返回编辑控件的格式矩形。

所谓格式矩形就是编辑控件中用于包含文本的范围,它与编辑控件窗口的尺寸无关。

- GetSel

调用该函数以得到编辑控件中当前被选文本的起始和终止字符位置,其原型为:

```
DWORD GetSel( ) const;  
void GetSel( int& nStartChar, int& nEndChar ) const;
```

返回值:

DWORD 版本函数的返回值中,低位字为被选文本的起始位置,而高位字为被选文本后第一个未选字符的位置。

参数:

nStartChar —— 将返回被选文本的起始位置。

nEndChar —— 将返回被选文本的终止位置。

- GetHandle

调用该函数以得到多行编辑控件的内存句柄,其原型为:

```
HLOCAL GetHandle( ) const;
```

返回值:

如果函数调用成功,则返回包含编辑控件内容的内存句柄,否则返回零值。

由 GetHandle 函数得到的句柄为本机内存句柄,并能在本地的任何 Windows 内存函数中作为参数被使用。

注意 如果对对话框中的编辑控件调用 GetHandle 函数,则该对话框必须具有 DS_LO-

CALEDIT 风格。否则,该函数将返回 0。并且 GetHandle 只能在 Windows NT 3.51 以后的版本中使用,而不能应用于 Windows 95。

- SetHandle

调用该函数以设置多行编辑控件的本地内存句柄,其原型为:

```
void SetHandle( HLOCAL hBuffer );
```

参数:

hBuffer —— 指定了本地内存句柄。该句柄必须由 LocalAlloc 函数创建,并指定 LMEM_MOVEABLE 标志。这块内存中将包含一个以空字符结尾的字符串,否则必须将这块内存中的第一个字节设置为 0。

调用了 SetHandle 函数后,编辑控件将使用由该函数分配的内存放置当前显示的文本,而不是自己分配内存来存放。

在设置新的内存句柄前,应该调用 GetHandle 成员函数以得到当前内存缓冲区的句柄,并调用 LocalFree 函数将其释放。

SetHandle 函数将清除“恢复(undo)”缓冲区,并将内部修改标志清除,然后重新绘制编辑框窗口中的内容。

注意 如果对对话框中的编辑控件调用 SetHandle 函数,则该对话框必须具有 DS_LOCALEEDIT 风格。否则,该函数将返回 0。并且 SetHandle 只能在 Windows NT 3.51 以后的版本中使用,而不能应用于 Windows 95。

- SetMargins

调用该函数以设置编辑控件中的左、右页边距,其原型为:

```
void SetMargins( UINT nLeft, UINT nRight );
```

参数:

nLeft —— 指定了新的左页边距,以像素为单位。

nRight —— 指定了新的右页边距,以像素为单位。

该成员函数既可以用于 Windows 9x/2000,也可以用于 Windows NT 4.0 以后版本。

- GetMargins

调用该函数以得到编辑控件的左、右页边距,其原型为:

```
DWORD GetMargins( ) const;
```

返回值:

如果函数调用成功则返回一个 DWORD 值,其高位字为右页边距,低位字为左页边距。该成员函数既可以用于 Windows 9x/2000,也可以用于 Windows NT 4.0 以后版本。

- SetLimitText

调用该函数以设置编辑控件可以包含的文本长度,其原型为:

```
void SetLimitText( UINT nMax );
```

参数:

nMax —— 指定了文本的字节长度。

该函数只能限制用户能输入的文本长度,它对已经存在于控件中的文本没有影响,同理它对于拷贝到控件中的文本(CWnd::SetWindowText)也没有影响。如果应用程序调用 CWnd::SetWindowText 函数,在编辑框中设置了超出 SetLimitText 所设置的文本范围,用户可以简单地删除他们。

- GetLimitText

调用该函数以得到编辑控件允许用户输入的最大文本长度,其原型为:

```
UINT GetLimitText( ) const;
```

返回值:

如果函数调用成功,则返回编辑控件中允许用户输入的最大文本长度。

该成员函数既可以用于 Windows 9x/2000,也可以用于 Windows NT 4.0 以后版本。

- PosFromChar

调用该函数以得到编辑控件中指定字符的左上角坐标,其原型为:

```
CPoint PosFromChar( UINT nChar ) const;
```

返回值:

如果函数调用成功,则返回由 nChar 指定的字符的左上角坐标。

参数:

nChar —— 指定了从零开始的字符索引。

使用 PosFromChar 函数可以得到编辑控件中指定字符的位置。如果 nChar 大于编辑控件中最后一个字符的索引,则返回值为编辑控件中最后一个字符后第一个像素的坐标。该成员函数既可以用于 Windows 9x/2000,也可以用于 Windows NT 4.0 以后版本。

- CharFromPos

调用该函数以得到最靠近指定位置处的字符索引和行号,其原型为:

```
int CharFromPos( CPoint pt ) const;
```

返回值:

如果函数调用成功,则返回值为 DWORD 值,其高位字为行号,而低位字为字符索引。

参数:

pt —— 指定了 CEdit 对象客户区中的一个位置。

该成员函数既可以用于 Windows 9x/2000,也可以用于 Windows NT 4.0 以后版本。

- GetLine

调用该函数以得到编辑控件中的一行文本,其原型为:

```
int GetLine( int nIndex, LPTSTR lpszBuffer ) const;  
int GetLine( int nIndex, LPTSTR lpszBuffer, int nMaxLength ) const;
```

返回值:

如果函数调用成功,则返回实际得到的字节数目。如果返回值为 0,则表示由 Index

指定的行号大于编辑控件的最后一行。

参数:

nIndex —— 指定了将从多行编辑控件中获取的行的行号。行号从零开始,也就是说该参数为 0 时,它表示编辑控件中的第一行。如果编辑控件为单行编辑控件,则该参数被忽略。

lpszBuffer —— 指定了将返回所获取的文本的缓冲区。该缓冲区中的第一个字(两个字节)指定了拷贝入缓冲区中的字节数目。

nMaxLength —— 指定了将拷贝到缓冲区中的最大字节数。在向 Windows 发送对该函数的调用之前,GetLine 函数将该值放置到 lpszBuffer 参数的第一个字中。

由 GetLine 函数获取的行中不包括终止字符。

- **GetPasswordChar**

当用户输入文本时,调用该函数以得到显示于编辑控件中的密码字符,其原型为:

```
TCHAR GetPasswordChar( ) const;
```

返回值:

如果函数调用成功,则返回将代替用户输入的密码字符,并显示于编辑控件中。如果返回值为空,则表示不存在密码字符。

如果编辑控件具有 ES_PASSWORD 风格,则默认的密码字符被设置为 *。

- **GetFirstVisibleLine**

调用该函数以得到编辑控件中的顶端可见行,其原型为:

```
int GetFirstVisibleLine( ) const;
```

返回值:

如果函数调用成功,则返回顶端可见行的行号,对于单行编辑控件,返回值总为零。

3.1.4 常规操作函数

CEdit 类的常规操作函数包括: EmptyUndoBuffer、FmtLines、LimitText、LineFromChar、LineIndex、LineLength、LineScroll、ReplaceSel、SetPasswordChar、SetRect、SetRectNP、SetSel、SetTabStops 和 SetReadOnly,它们能够完成清空编辑控件的恢复缓冲区、得到编辑控件中的被选文本等操作。

- **EmptyUndoBuffer**

调用该函数以清除编辑控件的恢复操作标志,其原型为:

```
void EmptyUndoBuffer( );
```

调用 EmptyUndoBuffer 函数后,编辑控件将不能再恢复上一次操作。当调用 SetWindowText 和 SetHandle 函数时,恢复操作标志会被自动清除。

- **FmtLines**

调用该函数以决定是否在多行编辑控件中插入软分行符,其原型为:

```
BOOL FmtLines( BOOL bAddEOL );
```

返回值:

如果没有任何格式化,则返回非零值,否则返回零值。

参数:

bAddEOL —— 指定了是否需要插入软分行符。如果该参数为 **TRUE**,则表示插入,否则表示去除。

软分行符实际包含两个回车和换行符,它的作用是将过长的行分成两行。而硬分行符则只包括一个回车和换行符。以硬分行符结尾的行不受 **FmtLines** 函数的影响。

需要注意的是,该函数只对多行编辑控件有效。并且它只对由 **GetHandle** 函数返回的缓冲区和由 **WM_GETTEXT** 返回的文本有效,而对编辑控件中显示的文本没有任何影响。

- **LimitText**

调用该函数以限制用户可能输入编辑控件中的文本长度,其原型为:

```
void LimitText( int nChars = 0 );
```

参数:

nChars —— 指定了用户可以输入的文本字节长度。如果该参数为 0,则可输入的文本长度被设置为 **UINT_MAX** 字节(默认值)。

该函数只对用户的输入有效,而对于已经存在于控件中的文本,或拷贝到控件中的文本无效。在 Windows NT 和 Windows 9x/2000 中, **SetLimitText** 代替了该函数。

- **LineFromChar**

调用该函数以得到指定字符索引所在的行,其原型为:

```
int LineFromChar( int nIndex = -1 ) const;
```

返回值:

如果函数调用成功,则返回包含指定字符索引的行号。如果 **nIndex** 为 -1,则返回包含当前被选文本的第一个字符的行;如果没有被选文本,则返回当前行号。

参数:

nIndex —— 指定了将获取其行号的字符索引。

该函数只对多行编辑控件有效。

- **LineIndex**

调用该函数以得到多行编辑控件中某行的字符索引,其原型为:

```
int LineIndex( int nLine = -1 ) const;
```

返回值:

如果函数调用成功,则返回 **nLine** 所指定的行的字符索引。如果返回值为 -1,则表示 **nLine** 大于编辑控件中的最后一行。

参数:

nLine —— 指定了将获取其索引的行号,如果该参数为 -1,则表示要得到的是当前行的索引。

所谓字符索引即某行的起始字符的索引,该函数只对多行编辑控件有效。

- **LineLength**

调用该函数以得到多行编辑控件中的行长度,其原型为:

```
int LineLength( int nLine = -1 ) const;
```

返回值:

如果对多行编辑控件调用该函数,则返回值为由 `nLine` 指定的行的字节长度。如果对单行编辑控件调用该函数,则返回值为编辑控件中的文本字节长度。

参数:

`nLine` —— 指定了将获取其所在行长度的字符索引。如果该参数为 `-1`,则函数得到的是当前行的长度(不包括该行中的被选文本)。如果对单行编辑控件调用该函数,则该参数被忽略。

其中 `nLine` 可以通过调用 `LineIndex` 函数得到。

- **LineScroll**

调用该函数以滚动多行编辑控件中的文本,其原型为:

```
void LineScroll( int nLines, int nChars = 0 );
```

参数:

`nLines` —— 指定了将垂直滚动的行数。

`nChars` —— 指定了将水平滚动到的字符位置。对于具有 `ES_RIGHT` 或 `ES_CENTER` 风格的编辑控件,该参数被忽略。

编辑控件不能垂直滚动过最后一行。如果当前行加上 `nLines` 参数所得值,超过了编辑控件中的总行数,则编辑控件的最后一行被滚动到窗口的最顶端。`LineScroll` 可以水平滚动过任何一行的最后一个字符。

- **ReplaceSel**

调用该函数以使用指定文本替换编辑控件中的当前被选文本,其原型为:

```
void ReplaceSel( LPCTSTR lpszNewText, BOOL bCanUndo = FALSE );
```

参数:

`lpszNewText` —— 指定了将用于替换被选文本的字符串。

`bCanUndo` —— 指定了替换操作是否可以被恢复。该参数的默认值为 `FALSE`。

该函数只能替换编辑控件中的部分文本,如果要替换其中的全部文本则需要调用 `SetWindowText` 函数。如果当前没有被选文本,则将替换文本插入当前的输入位置。

- **SetPasswordChar**

调用该函数以设置或去除显示在编辑控件中的密码字符,其原型为:

```
void SetPasswordChar( TCHAR ch );
```

参数:

`ch` —— 指定了将用于替代用户输入而显示在编辑框中的字符。如果该参数为 `0`,则显示用户输入的实际字符。

如果设置 `ch` 为非零值,则用户输入的每个字符都会以 `ch` 的形式显示。该函数对于多行编辑控件无效。当调用 `SetPasswordChar` 函数时, `CEdit` 将所有可见字符都使用 `ch` 重新绘制。

如果编辑控件具有 `ES_PASSWORD` 风格,则默认密码字符为 `*`。如果在调用 `SetPasswordChar` 函数时,将 `ch` 设置为 `0`,则可以去除编辑控件的这个风格。

- **SetRect**

调用该函数以设置多行编辑控件的格式矩形,并更新控件,其原型为:

```
void SetRect( LPCRECT lpRect );
```

参数:

`lpRect` —— 指定了格式矩形的新尺寸,该参数可以为 `RECT` 结构或 `CRect` 对象。

该函数只对多行编辑控件有效。格式矩形就是文本的限制矩形,它与编辑控件窗口的尺寸无关。当编辑控件被创建时,格式矩形与编辑控件的客户区窗口尺寸一致。通过 `SetRect` 函数,应用程序可以使格式矩形大于或小于编辑控件窗口。

如果编辑控件没有滚动条,而格式矩形又大于控件窗口,则文本将被裁剪,而不是被换行。如果编辑控件包括边框,则格式矩形将被限制在边框的范围内。如果调整了由 `GetRect` 函数返回的矩形,则在将修改后的矩形发送给 `SetRect` 函数前,从中去除边框的尺寸。

调用 `SetRect` 函数后,编辑控件将被重新格式化,并重新显示。

- **SetRectNP**

调用该函数以设置多行编辑控件的格式矩形,但不重绘编辑控件窗口,其原型为:

```
void SetRectNP( LPCRECT lpRect );
```

参数:

`lpRect` —— 指定了格式矩形的新尺寸,该参数可以为 `RECT` 结构或 `CRect` 对象。

`SetRectNP` 与 `SetRect` 函数完全一致,只是对前者的调用不会导致控件窗口的重绘。

- **SetSel**

调用该函数以设置编辑控件中的当前被选文本,其原型为:

```
void SetSel( DWORD dwSelection, BOOL bNoScroll = FALSE );  
void SetSel( int nStartChar, int nEndChar, BOOL bNoScroll = FALSE );
```

参数:

`dwSelection` —— 指定了被选文本的范围,参数的低位字为其终止位置,高位字为其起始位置。如果低位字为 `0`,而高位字为 `-1`,则编辑控件中的所有文本都被选中。如果低位字为 `-1`,则任何当前的被选文本都被去除。

`bNoScroll` —— 指定了是否应该滚动编辑控件,以使被选文本可见。如果该参数为 `FALSE`,则将滚动以使被选文本可见,否则将不滚动编辑控件。

`nStartChar` —— 指定了被选文本的起始位置。如果 `nStartChar` 为 `0`,而 `nEndChar` 为 `-1`,则编辑控件中的所有文本都被选中。如果 `nStartChar` 为 `-1`,则将去除编辑控件中

的所有被选文本。

nEndChar —— 指定了被选文本的终止位置。

- SetTabStops

调用该函数以设置多行文本中的 tab 间隔,其原型为:

```
void SetTabStops( );  
BOOL SetTabStops( const int& cxEachStop );  
BOOL SetTabStops( int nTabStops, LPINT rgTabStops );
```

返回值:

如果设置了 tab,则返回非零值,否则返回零值。

参数:

cxEachStop —— 指定了对于每个 cxEachStop 对话框单位,都设置 tab 间隔。

nTabStops —— 指定了 rgTabStops 中包含的 tab 间隔的数目,该参数必须大于 1。

rgTabStops —— 指定了存放 tab 间隔的无符号整数数组,其中 tab 间隔以对话框单位计算。对话框单位为一个垂直或水平的距离。水平对话框单位等于当前对话框长度单位的 1/4,而垂直对话框单位则等于当前对话框高度单位的 1/8。对话框单位是根据当前系统字体的高度和宽度计算得出的。调用 GetDialogBaseUnits 函数 可以得到对话框单位的数值。

当文本被拷贝到多行编辑控件中时,其中的任何 tab 字符都会导致生成间隔。

调用无参数版本的 SetTabStops 函数将 tab 间隔设置为默认的 32 倍对话框单位。要设置 tab 间隔为其他值,则需要调用带有 cxEachStop 参数的 SetTabStops 函数版本。如果要设置间隔为一组尺寸,则需要使用带有两个参数的 SetTabStops 函数版本。

该函数只对多行编辑控件有效,并且它不会自动重绘编辑控件窗口,因此需要调用 InvalidateRect 函数以重绘窗口。

- SetReadOnly

调用该函数以设置编辑控件是否只读,其原型为:

```
BOOL SetReadOnly( BOOL bReadOnly = TRUE );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

bReadOnly —— 指定了编辑控件是否将被设置为只读。该参数为 FALSE,则表示编辑控件非只读,否则编辑控件为只读。

3.1.5 剪贴板操作

CEdit 类的剪贴板操作函数包括: Undo、Clear、Copy、Cut 和 Paste,它们能够完成与编辑控件相关的恢复、拷贝、剪贴等剪贴板操作。

- Undo

调用该函数以恢复编辑控件的上一个操作,其原型为:

```
BOOL Undo( );
```

返回值:

对于单行编辑控件返回值总为零值。而对于多行编辑控件,如果恢复操作成功完成,则返回非零值,否则返回零值。

恢复操作也可以被撤消。例如调用 Undo 恢复被删除的文本后,紧接着再次调用可以撤消对被删除文本的恢复。

- Clear

调用该函数以删除编辑控件中的当前被选文本,其原型为:

```
void Clear( );
```

Clear 对被选文本的删除,可以通过 Undo 函数恢复。

- Copy

调用该函数以将当前编辑控件中的被选文本,以 CF_TEXT 格式拷贝到剪贴板中,其原型为:

```
void Copy( );
```

- Cut

调用该函数以删除当前编辑框中的被选文本,并将其以 CF_TEXT 格式拷贝到剪贴板中,其原型为:

```
void Cut( );
```

Cut 对被选文本的删除,可以通过 Undo 函数恢复。

- Paste

调用该函数可以将剪贴板中的数据,插入编辑控件中的当前位置(这只有在剪贴板中的数据为 CF_TEXT 格式时才行)。Paste 函数的原型为:

```
void Paste( );
```

3.2 CEditView 类

CEditView 对象与 CEdit 对象类似,但它除了为 Windows 编辑控件提供支持外,还能够实现简单地文本编辑器功能。图 3-2 所示为 CEditView 类的派生结构。相对于 CEdit 类,CEditView 类提供了打印、寻找和替换等附加功能。

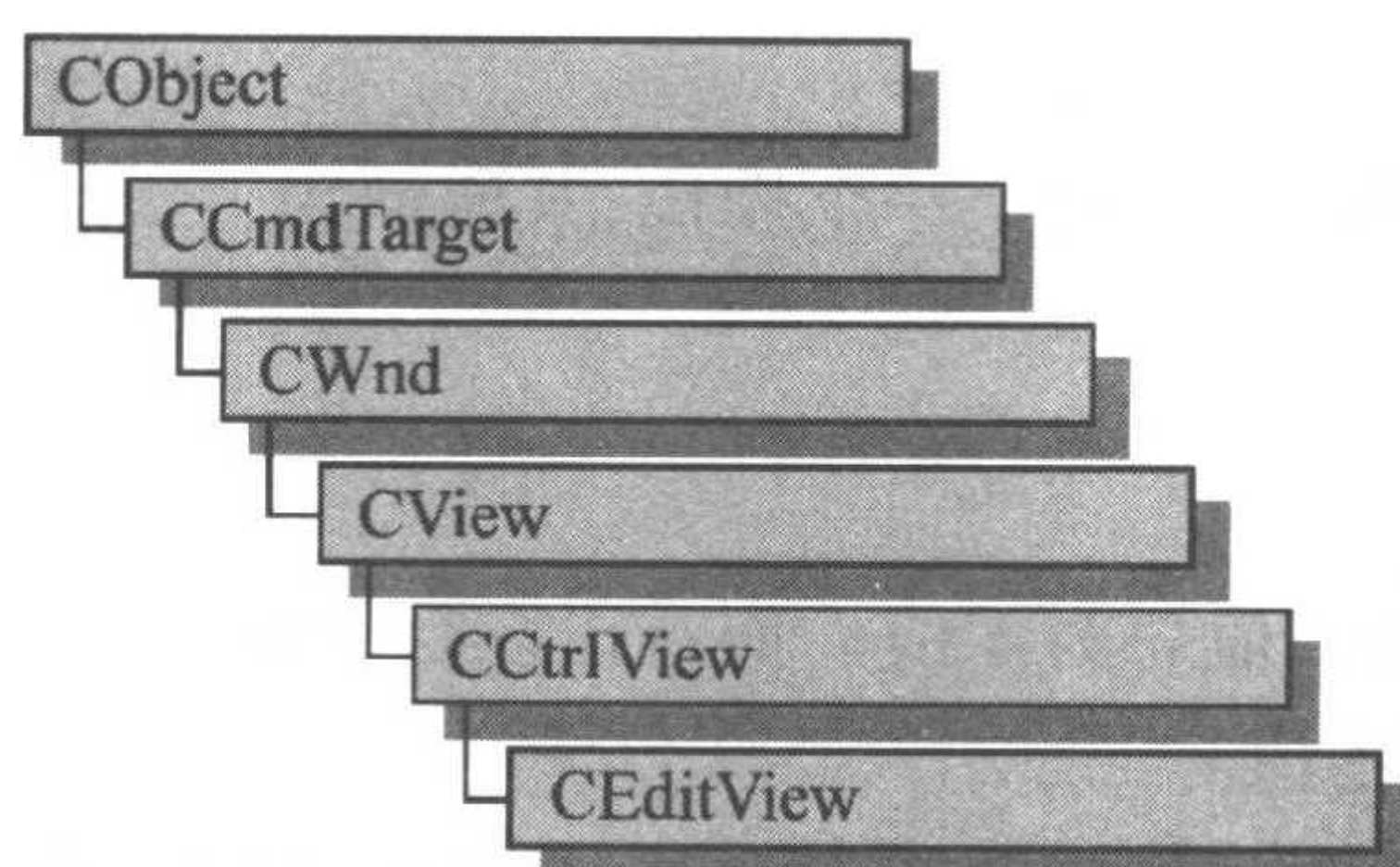


图 3-2 CEditView 类的派生结构

3.2.1 CEditView 类概述

由于 CEditView 是 CView 的派生类,因此 CEditView 对象可以用于文档/视图结构的应用程序。CEditView 控件中的文本由控件本身的全局内存对象保存。应用程序中可以有多个 CEditView 对象。

当希望使用具有上述附加功能的编辑窗口,或需要简单文本编辑器时,可以考虑创建 CEditView 对象。CEditView 对象能够占据整个窗口客户区。如需扩展 CEditView 类的功能,则可以创建其派生类。

在默认情况下,CEditView 类将处理如下命令: ID_EDIT_SELECT_ALL、ID_EDIT_FIND、ID_EDIT_REPLACE、ID_EDIT_REPEAT 和 ID_FILE_PRINT。

不过 CEditView 及其派生类具有如下缺点:

- CEditView 没有实现 WYSIWYG(所见即所得)编辑。
- CEditView 只能以一种字体显示文本,并且不支持任何特殊字符格式。
- CEditView 对象中能够包含的文本是有限的,其上限与 CEdit 控件一致。

3.2.2 构造函数

CEditView 类的构造函数为: CEditView, 调用该函数以构造 CEditView 对象,其原型如下:

```
CEditView( );
```

在构造了 CEditView 对象后,必须在使用它之前调用 Create 函数。如果使用 CEditView 派生类,并调用 CWinApp::AddDocTemplate 将其添加到模板中,框架将负责调用构造和 Create 函数。

3.2.3 属性操作函数

CEditView 类的属性操作函数包括: GetEditCtrl、GetPrinterFont、GetSelectedText、Lock-

Buffer、UnlockBuffer、GetBufferLength、SetPrinterFont 和 SetTabStops, 它们能够完成得到 CEdit 对象、锁定缓冲区等操作。

- **GetEditCtrl**

调用该函数以提供对 CEditView 对象中的 CEdit 部分的存取, 其原型为:

```
CEdit& GetEditCtrl() const;
```

返回值:

如果函数调用成功, 则返回 CEdit 引用。

通过 CEdit 引用可以直接使用 Windows 编辑控件 CEdit 的成员函数。不过需要注意的是, 使用 CEdit 对象可能改变底层 Windows 编辑控件。例如, 不应该使用 CEdit::SetTabStops 函数改变 tab 设置, 而应该使用 CEditView::SetTabStops 函数。这是因为 CEditView 将缓存这些设置, 并将其用在编辑控件和打印中。

- **GetPrinterFont**

调用该函数以得到当前打印机字体, 其原型为:

```
CFont * GetPrinterFont() const;
```

返回值:

如果函数调用成功, 则返回指定打印机字体的 CFont 对象指针。如果打印机还没有被设置, 则返回 NULL。

- **GetSelectedText**

调用该函数以得到当前被选中的文本, 其原型为:

```
void GetSelectedText( CString& strResult ) const;
```

参数:

strResult —— 将返回被选中的文本。

- **LockBuffer**

调用该函数以锁定缓冲区, 其原型如下:

```
LPCTSTR LockBuffer() const;
```

返回值:

如果函数调用成功, 则返回编辑控件缓冲区的指针。

- **UnlockBuffer**

调用该函数以解除对缓冲区的锁定, 其原型为:

```
void UnlockBuffer() const;
```

- **GetBufferLength**

调用该函数以得到字符缓冲区的长度, 其原型为:

```
UINT GetBufferLength() const;
```


返回值:

如果函数调用成功,则返回字符缓冲区的长度。

- **SetPrinterFont**

调用该函数以设置新的打印机字体,其原型为:

```
void SetPrinterFont( CFont * pFont );
```

参数:

pFont —— 指定了将使用的新字体。如果该参数为 NULL,则打印机将使用显示字体。

如果希望使用特殊的字体进行打印,则应该在类的 OnPreparePrinting 函数中调用 SetPrinterFont。该虚函数必须在打印前调用,这样就能够保证在打印前改变字体。

- **SetTabStops**

调用该函数以设置屏幕显示和打印的制表符,其原型为:

```
void SetTabStops( int nTabStops );
```

参数:

nTabStops —— 指定了制表符的宽度。

CEditView 只支持单制表符宽度(CEdit 支持多制表符宽度),其宽度单位等于打印或显示时使用的字体的 1/4 平均字符宽度。不过一定不要使用 CEdit::SetTabStops。该函数只修改调用它的 CEditView 对象的制表符宽度。

3.2.4 常规操作函数

CEditView 类的常规操作函数包括: FindText、PrintInsideRect 和 SerializeRaw 函数,它们能够完成检索文本、保存 CEditView 对象等操作。

- **FindText**

调用该函数以在文本中检索指定文本,其原型为:

```
BOOL FindText( LPCTSTR lpszFind, BOOL bNext = TRUE, BOOL bCase = TRUE );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

lpszFind —— 指定了将检索的文本。

bNext —— 指定了检索方向。如果该参数为 TRUE,则检索方向指向缓冲区的末端。如果该参数为 FALSE,则检索方向指向缓冲区的开头。

bCase —— 指定了检索是否是大小写敏感的。如果该参数为 TRUE,则区分字符的大小写。而如果该参数为 FALSE,则检索不区分字符的大小写。

- **PrintInsideRect**

调用该函数以打印指定矩形中的文本,其原型为:

```
UINT PrintInsideRect( CDC * pDC, RECT& rectLayout, UINT nIndexStart, UINT nIndexEnd );
```

dexStop);

返回值:

如果函数调用成功,则返回将打印的下一个字符的索引。

参数:

pDC —— 指定了打印机设备环境。

rectLayout —— 指定了包含打印文本的矩形,可以为 CRect 对象也可以为 RECT 结构。

nIndexStart —— 指定了缓冲区中将被打印的第一个字符的索引。

nIndexStop —— 指定了缓冲区中最后一个被打印字符后的字符索引。

如果 CEditView 控件没有 ES_AUTOHSCROLL 风格,则文本将在矩形中回绕。如果控件具有 ES_AUTOHSCROLL 风格,则文本将被矩形的右边界裁剪。

- SerializeRaw

调用该函数以将 CEditView 对象以纯文本方式存储在磁盘中,其原型为:

```
void SerializeRaw( CArchive& ar );
```

参数:

ar —— 指定了将用以序列化文本的 CArchive 对象。

SerializeRaw 函数与 CEditView 内部实现的 Serialize 函数不同,该函数只读取和写入文本,而不能处理其他类型的数据。

3.2.5 重载函数

CEditView 类的重载函数包括: OnFindNext、OnReplaceAll、OnReplaceSel 和 OnTextNotFound,它们能够完成替换文本等操作。

- OnFindNext

调用该函数以在文本缓冲区中检索指定文本,其原型为:

```
virtual void OnFindNext( LPCTSTR lpszFind, BOOL bNext, BOOL bCase );
```

参数:

lpszFind —— 指定了将被检索的文本。

bNext —— 指定了检索方向。如果该参数为 TRUE,则检索方向指向缓冲区的末端。如果该参数为 FALSE,则检索方向指向缓冲区的开头。

bCase —— 指定了检索是否是大小写敏感的。如果该参数为 TRUE,则区分字符的大小写。而如果该参数为 FALSE,则检索不区分字符的大小写。

检索操作将从当前被选文本开始,而通过对 FindText 函数的调用完成。在默认情况下,如果文本没有找到,则 OnFindNext 会调用 OnTextNotFound 函数。

如果希望改变文本的检索方式,则应该重载 OnFindNext 函数。CEditView 将在用户单击标准“查找和替换”对话框中的“查找下一处”按钮时,调用 OnFindNext 函数。

- OnReplaceAll

调用该函数以使用指定文本替换 CEditView 中的所有文本,其原型为:

```
virtual void OnReplaceAll( LPCTSTR lpszFind, LPCTSTR lpszReplace, BOOL bCase );
```

参数:

lpszFind —— 指定了将检索的文本。

lpszReplace —— 指定了将用于替换的文本。

bCase —— 指定了检索是否是大小写敏感的。如果该参数为 TRUE,则区分字符的大小写。而如果该参数为 FALSE,则检索不区分字符的大小写。

当用户按下标准“查找与替换”对话框中的“替换全部”按钮时,CEditView 调用 OnReplaceAll 函数。如果希望改变文本的替换方式,则应该重载 OnReplaceAll 函数。

- OnReplaceSel

调用该函数以替换当前被选文本,其原型为:

```
virtual void OnReplaceSel( LPCTSTR lpszFind, BOOL bNext, BOOL bCase, LPCTSTR lpszReplace );
```

参数:

lpszFind —— 指定了将检索的文本。

bNext —— 指定了检索方向。如果该参数为 TRUE,则检索方向指向缓冲区的末端。如果该参数为 FALSE,则检索方向指向缓冲区的开头。

bCase —— 指定了检索是否是大小写敏感的。如果该参数为 TRUE,则区分字符的大小写。而如果该参数为 FALSE,则检索不区分字符的大小写。

lpszReplace —— 指定了将用于替换的文本。

当用户按下标准“查找与替换”对话框中的“替换”按钮时,CEditView 调用 OnReplaceSel 函数。如果希望改变文本的替换方式,则应该重载 OnReplaceSel 函数。

- OnTextNotFound

调用该函数以响应检索失败事件,其原型为:

```
virtual void OnTextNotFound( LPCTSTR lpszFind );
```

参数:

lpszFind —— 指定了将检索的文本。

默认情况下,该函数将调用 Windows 的 MessageBeep 函数以发出响声。

3.3 改变控件的外观

3.3.1 能够保持“高亮”状态的编辑控件

读者在设计应用程序或使用应用程序时,可能遇到过界面上有很多需要输入数据的编辑框的情况。有时可能出现的这样的情况:当用户在某个编辑框中输入数据时,需要切换到另一个应用程序(例如 FTP 下载数据完毕)进行处理。而当用户再切换回原来的输

入口时,需要费心确定刚刚是在哪个编辑框中输入数据,因为此时所有的编辑控件看起来都是一,并没有明显的证据表明哪个编辑框是用户最近操作的。

那么如何解决这个问题呢?实际上,我们可以考虑列表视图控件等采用的方法:当父窗口失去焦点然后又得到焦点时,被选中的条目依然会保持它的高亮状态(不同的背景)。对于编辑控件当然不必一定使用不同的背景颜色,最简单的办法就是当编辑控件失去或得到焦点时,改变其边界形状(例如使其变为 3D 外观)。而这从某种程度上,这也相当于“高亮”,同时也能帮助用户很容易地定位具有输入焦点的编辑框。

下面创建了一个 CBorderEdit 类,它由 CEdit 派生。在其中添加了对 WM_SETFOCUS 和 WM_KILLFOCUS 消息(得到和失去输入焦点)的处理函数,在其中调用 CWnd::ModifyStyleEx 函数修改编辑控件的风格。清单 3-1 和 3-2 所示为 OnSetFocus 和 OnKillFocus 函数的源代码:

清单 3-1 OnSetFocus()函数

```
void CBorderEdit::OnSetFocus(CWnd* pOldWnd)
{
    CEdit::OnSetFocus(pOldWnd);
    ModifyStyleEx(0, WS_EX_DLGMODALFRAME | WS_EX_WINDOWEDGE,
        SWP_FRAMECHANGED);
}
```

清单 3-2 OnKillFocus()函数

```
void CBorderEdit::OnKillFocus(CWnd* pNewWnd)
{
    CEdit::OnKillFocus(pNewWnd);
    ModifyStyleEx(WS_EX_DLGMODALFRAME | WS_EX_WINDOWEDGE, 0,
        SWP_FRAMECHANGED);
}
```

在使用时,只要使编辑控件由 CBorderEdit 类管理即可(例如,通过 ClassWizard 为编辑控件添加 CBorderEdit 类型的变量)。图 3-3 所示即为在父窗口重新得到焦点时,也能保持自己的“高亮”状态的编辑框。



图 3-3 能够保持“高亮”状态的编辑框

3.3.2 鼠标敏感编辑控件

大家可能都很喜欢那种平时具有“平坦”外观,而当鼠标经过其上就变成 3D 外观的控件。在本节中就向读者介绍如何创建鼠标敏感的编辑控件。

实际上无论是编辑控件还是按钮控件的外观,都是由其边框的绘制方法决定的(这在第 2 章中已经有过介绍)。平坦外观的编辑控件没有绘制边框,而 3D 外观的编辑控件绘制 3D 边框。下面将创建一个 CHotEdit 类(CEdit 的派生类),它将负责管理编辑控件边框的绘制。清单 3-3 所示为 CHotEdit 类的定义:

清单 3-3 CHotEdit 类的定义

```
class CHotEdit : public CEdit
{
// Construction
public:
    CHotEdit();

// Implementation
public:
    virtual ~CHotEdit();

    // Generated message map functions
protected:
    virtual void DrawBorder(bool fHot = true);
    COLORREF m_clr3DHilight;
    COLORREF m_clr3DLight;
    COLORREF m_clr3DDkShadow;
    COLORREF m_clr3DShadow;
    COLORREF m_clr3DFace;
    bool m_fTimerSet;
    bool m_fGotFocus;
    //{AFX_MSG(CHotEdit)
    afx_msg void OnPaint();
    afx_msg void OnSetFocus(CWnd* pOldWnd);
    afx_msg void OnKillFocus(CWnd* pNewWnd);
    afx_msg void OnMouseMove(UINT nFlags, CPoint point);
    afx_msg void OnTimer(UINT nIDEvent);
    afx_msg void OnNcMouseMove(UINT nHitTest, CPoint point);
    afx_msg void OnSysColorChange();
    //}AFX_MSG

    DECLARE_MESSAGE_MAP()
};
```

其中 COLORREF 类型的变量用于设置不同的边框颜色;m_fTimerSet 用于标识是否设置了定时器;m_fGotFocus 用于标识控件是否具有焦点。

由于控件边框的绘制是由鼠标动作引起的,因此显然要在类中处理鼠标移动消息,以

设置控件边框显示与否。清单 3-4 所示为 OnMouseMove 函数的源代码：

清单 3-4 OnMouseMove() 函数

```
void CHotEdit::OnMouseMove(UINT nFlags, CPoint point)
{
    if (! m_fGotFocus) {
        // 如果没有设置定时器,则设置它
        if (! m_fTimerSet) {
            DrawBorder();
            SetTimer(1, 10, NULL);
            m_fTimerSet = true;
        }
    }

    CEdit::OnMouseMove(nFlags, point);
}
```

如果控件具有输入焦点,当然控件的外观不需改变。因此在上述代码中,首先检查控件是否具有输入焦点。如果控件没有输入焦点,则根据鼠标位置绘制按钮边框。读者也许已经发现,OnMouseMove 函数中看不出有判断鼠标位置的代码,这实际上是通过定时器响应函数完成的。清单 3-5 所示为 OnTimer 函数：

清单 3-5 OnTimer() 函数

```
void CHotEdit::OnTimer(UINT nIDEvent)
{
    POINT pt;
    GetCursorPos(&pt);
    CRect rcItem;
    GetWindowRect(&rcItem);

    // 鼠标是否在控件之上?
    if(! rcItem.PtInRect(pt)) {
        KillTimer(1);

        m_fTimerSet = false;

        if (! m_fGotFocus) {
            DrawBorder(false);
        }

        return;
    }

    CEdit::OnTimer(nIDEvent);
}
```

当然,判断鼠标位置的操作也可以在 OnMouseMove 函数中完成,这与程序员的喜好有关。除了鼠标位置会影响控件外观外,控件是否具有焦点也会影响其外观。因此还应该处理 WM_SETFOCUS 和 WM_KILLFOCUS 消息。清单 3-6 和 3-7 所示为 OnSetFocus 和 OnKillFocus 函数的源代码：

清单 3-6 OnSetFocus() 函数

```
void CBorderEdit::OnSetFocus(CWnd * pOldWnd)
{
    CEdit::OnSetFocus(pOldWnd);
    m_fGotFocus = true;
    DrawBorder();
}
```

清单 3-7 OnKillFocus() 函数

```
void CHotEdit::OnKillFocus(CWnd * pNewWnd)
{
    CEdit::OnKillFocus(pNewWnd);
    m_fGotFocus = false;
    DrawBorder(false);
}
```

另外,边框绘制操作被封装在 DrawBorder 函数中,其源代码如清单 3-8 所示:

清单 3-8 DrawBorder() 函数

```
void CHotEdit::DrawBorder(bool fHot)
{
    CRect rcItem;
    DWORD dwExStyle = GetExStyle();
    CDC * pDC = GetDC();
    COLORREF clrBlack;
    int nBorderWidth = 0;
    int nLoop;

    GetWindowRect(&rcItem);
    ScreenToClient(&rcItem);

    clrBlack = RGB(0, 0, 0);

    if (! IsWindowEnabled()) {
        fHot = true;
    }

    if (dwExStyle & WS_EX_DLGMODALFRAME) {
        nBorderWidth += 3;
    }

    if (dwExStyle & WS_EX_CLIENTEDGE) {
        nBorderWidth += 2;
    }

    if (dwExStyle & WS_EX_STATICEDGE && ! (dwExStyle & WS_EX_DLGMODALFRAME)) {
        nBorderWidth ++;
    }

    for (nLoop = 0; nLoop < nBorderWidth; nLoop++) {
        pDC->Draw3dRect(rcItem, m_clr3DFace, m_clr3DFace);
        rcItem.DeflateRect(1, 1);
    }
}
```

```

}
rcItem.InflateRect(1, 1);

if (fHot) {
if (dwExStyle & WS_EX_CLIENTEDGE) {
pDC->Draw3dRect(rcItem, m_clr3DDkShadow, m_clr3DLight);
rcItem.InflateRect(1, 1);
pDC->Draw3dRect(rcItem, m_clr3DShadow, m_clr3DHilight);
rcItem.InflateRect(1, 1);
}

if (dwExStyle & WS_EX_STATICEDGE && ! (dwExStyle & WS_EX_DLGMODALFRAME)) {
pDC->Draw3dRect(rcItem, m_clr3DShadow, m_clr3DHilight);
rcItem.InflateRect(1, 1);
}

if (dwExStyle & WS_EX_DLGMODALFRAME) {
pDC->Draw3dRect(rcItem, m_clr3DFace, m_clr3DFace);
rcItem.InflateRect(1, 1);
pDC->Draw3dRect(rcItem, m_clr3DHilight, m_clr3DShadow);
rcItem.InflateRect(1, 1);
pDC->Draw3dRect(rcItem, m_clr3DLight, m_clr3DDkShadow);
}
}

ReleaseDC(pDC);
}

```

在使用时,只要使编辑控件由 CHotEdit 类管理即可(例如,通过 ClassWizard 为编辑控件添加 CHotEdit 类型的变量)。图 3-4 所示即为鼠标敏感编辑控件。

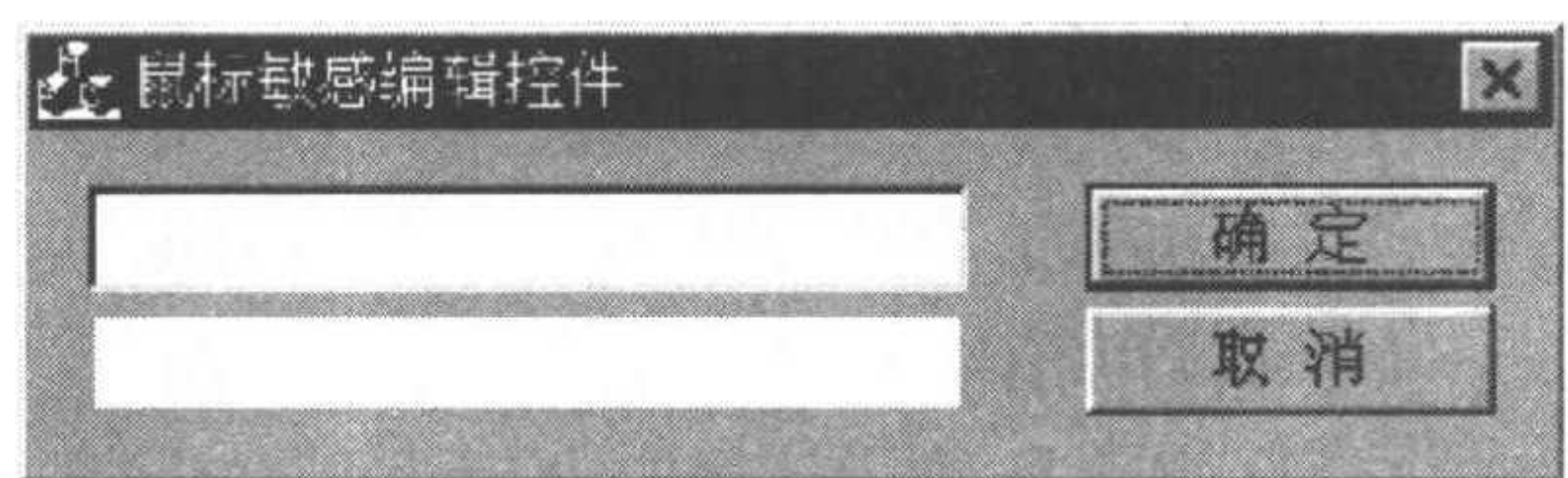


图 3-4 鼠标敏感编辑控件

3.4 改变控件的编辑及显示方式

3.4.1 限制输入的数据类型

编辑框的用途多种多样,其中可以接受多种数据的输入。但是这往往也会带来一些问题,例如如何使用户只能输入 1.23、12e4 或 -12E4 之类的双精度值,或是使用户只能进行十六进制编辑操作等。

1. 编辑双精度数值

限制编辑框中只能输入双精度数值,可能是最容易实现的一种方式了。按照编程的基本原则,首先应该创建一个用于管理双精度数值编辑框的类 CNumEdit,当然它应该是 CEdit 的派生类。为了使读者更好地理解下面将进行的工作,这里先向读者介绍一些键盘编程的基础知识。

• 扫描码和虚拟键码

键盘上的每一个键都对应着一个唯一的标识值,即扫描码。当用户按下或释放一个键时,都会产生相应的扫描码。虽然扫描码可以作为键的标识使用,但是它的一个缺陷在于其设备依赖性。因此,一般使用与设备无关的虚拟键码。所谓虚拟键码就是由 Windows 定义的与设备无关的键标识值,如表 3-4 所示。键盘消息的 wParam 参数中包含虚拟键码值。

表 3-4 虚拟键码

键盘键	虚拟键码	十六进制键值
退格键	VK_BACK	08
制表键(Tab 键)	VK_TAB	09
清除键(Clear 键)	VK_CLEAR	0C
回车键(Enter 键)	VK_RETURN	0D
Shift 键	VK_SHIFT	10
Ctrl 键	VK_CONTROL	11
Alt 键	VK_MENU	12
Pause 键	VK_PAUSE	13
Caps Lock 键	VK_CAPTIAL	14
Esc 键	VK_ESCAPE	1B
空格键	VK_SPACE	20
Page Up 键	VK_PRIOR	21
Page Down 键	VK_NEXT	22
End 键	VK_END	23
Home 键	VK_HOME	24
←键	VK_LEFT	25
↑键	VK_UP	26
→键	VK_RIGHT	27
↓键	VK_DOWN	28
Select 键	VK_SELECT	29
Execute 键	VK_EXECUTE	2B
Print Screen 键	VK_SNAPSHOT	2C
Insert 键	VK_INSERT	2D
Delete 键	VK_DELETE	2E
Help 键	VK_HELP	2F
0~9 键	VK_0 ~ VK_9	30 ~ 39

续表

键盘键	虚拟键码	十六进制键值
A ~ Z 键	VK_A ~ VK_Z	41 ~ 5A
小数字键盘的 0 ~ 9 键	VK_NUMPAD0 ~ VK_NUMPAD9	60 ~ 69
乘号键(* 键)	VK_MULTIPLY	6A
加号键(+ 键)	VK_ADD	6B
分隔符键(键)	VK_SEPARATOR	6C
减号键(- 键);	VK_SUBTRACT	6D
小数点键(. 键)	VK_DECIMAL	6E
除号键(/ 键)	VK_DIVIDE	6F
F1 ~ F24 键	VK_F1 ~ VK_F24	70H ~ 87H
Num Lock 键	VK_NUMLOCK	90
Scroll Lock 键	VK_SCROLL	91

当用户击键时,键盘驱动程序 Keyboard.drv 中的键盘中断程序都会对所击键进行编码,然后将其翻译成为虚拟键码。然后调用 Windows 的用户模块 User.exe 中的有关程序来生成键盘消息,在该消息中含有扫描码、虚拟键码以及其他与击键有关的信息。最后 Windows 从系统消息队列中将键盘消息取出并发送到相应的应用程序的消息队列中等待处理。而这些键盘消息的处理则由应用程序完成。

• 输入焦点

在 Windows 操作系统中,键盘一般由所有的应用程序所共享,不过也有例外。例如在 DirectInput 中允许将键盘设置为独占模式,有关内容在笔者的拙作《Visual C++ 6.0 多媒体开发指南》中有详细介绍。当按下键盘上的一个键时,只有一个应用程序窗口能够接收到该键盘消息,这个窗口就被称为具有“输入焦点”的窗口。具有输入焦点的窗口可以是活动窗口,也可以是活动窗口中的子窗口。如果活动窗口为一个图标,Windows 将仍向该窗口发送键盘消息,只是消息格式有所不同。当一个窗口获得输入焦点时,会发送 WM_SETFOCUS 消息;当其失去输入焦点时,会发送 WM_KILLFOCUS 消息。系统正是通过捕获这两个消息,来判断窗口是否具有输入焦点。

• 系统键和非系统键

系统键一般是由输入键和 Alt 键组合产生,这种输入一般由 Windows 内部直接处理,而应用程序则通常不需考虑。如果用户希望在应用程序中处理系统键,则在处理完毕后,应该调用 DefWindowProc 函数,以恢复 Windows 对其的默认处理。按下系统键会发送两条系统击键消息: WM_SYSKEYDOWN 和 WM_SYSKEYUP。非系统键则指不与 Alt 联合输入的键,按下非系统键会发送两条击键消息: WM_KEYDOWN 和 WM_KEYUP。

• 键盘消息映射函数

当用户按下一个键或组合键时,Windows 会将 WM_KEYDOWN 和 WM_KEYUP 消息发送给具有输入焦点的应用程序窗口进行处理。MFC 为这些键盘消息提供了消息处理虚函数,如表 3-5 所示。

表 3-5 键盘消息处理虚函数

键盘消息	含义	消息处理函数
WM_KEYDOWN	按下非系统键	OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
WM_KEYUP	释放非系统键	OnKeyUp(UINT nChar, UINT nRepCnt, UINT nFlags)
WM_SYSKEYDOWN	按下系统键	OnSysKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
WM_SYSKEYUP	释放系统键	OnSysKeyUp(UINT nChar, UINT nRepCnt, UINT nFlags)

键盘消息处理函数中的参数 nChar 记录了键的 OEM 扫描码。参数 nRepCnt 记录了击键的重复次数。参数 nFlags 记录了前一次击键的状态,其标志位含义如表 3-6 所示:

表 3-6 nFlags 参数的标志位

标志位	含义
0 ~ 7	包含键盘扫描码
8	如果为 1,则为增强键盘的扩展键
9 ~ 10	保留
11 ~ 12	Windows 内部使用
13	如果为 1,则同时按下了 Alt 键
14	前一次击键状态
15	如果为 1,则为按下键,否则为释放键

• 字符消息处理函数

当击键输入可显示字符时,Windows 在发送键盘消息的同时还发送字符消息。在 Windows 中共有 4 条字符消息,这些字符消息及其消息处理函数如表 3-7 所示:

表 3-7 字符消息处理函数

字符消息	消息处理函数
WM_CHAR	OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
WM_DEADCHAR	OnDeadChar(UINT nChar, UINT nRepCnt, UINT nFlags)
WM_SYSCHAR	OnSysChar(UINT nChar, UINT nRepCnt, UINT nFlags)
WM_SYSDEADCHAR	OnSysDeadChar(UINT nChar, UINT nRepCnt, UINT nFlags)

以上消息处理函数中的参数与键盘消息处理函数中的参数含义相同。由于击键方式和击键次序不同,窗口所接到的消息的数目和次序也不相同。例如对于按下 A 键然后释放的操作,窗口所接到的消息次序为: WM_KEYDOWN、WM_CHAR 和 WM_KEYUP。

由上所述,只要使编辑框只对数字键、e 或 E 键的输入作出响应,就可以使其中的输入限制于双精度数值。因此,在 CNumEdit 类中应该添加对 WM_CHAR 消息的响应函数。如清单 3-9 所示。

清单 3-9 OnChar()函数

```
void CNumEdit::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    // TODO: Add your message handler code here and/or call default
```

```
if (nChar == 8)
    CEdit::OnChar(nChar, nRepCnt, nFlags);

POINT caret;
::GetCaretPos (&caret);
caret.x = LOWORD (CharFromPos (caret));

CString text;
GetWindowText (text);

if (isdigit(nChar))
    CEdit::OnChar(nChar, nRepCnt, nFlags);
else if (nChar == '-')
{
    if (! caret.x)
    {
        if (((text.GetLength() > 0) && (text[0] != '-')) || (text.
            GetLength() == 0))
            CEdit::OnChar(nChar, nRepCnt, nFlags);
    }
    else
    {
        if ((text[caret.x-1] == 'e') || (text[caret.x-1] == 'E'))
            CEdit::OnChar(nChar, nRepCnt, nFlags);
    }
}

else if ((nChar == 'e') || (nChar == 'E'))
{
    if ((caret.x == 1) && (text[0] == '-'))
        return ;

    if (caret.x)
    {
        for (int i = 0; i < text.GetLength(); i++)
        {
            if ((text[i] == 'e') || (text[i] == 'E'))
                return ;
        }
        CEdit::OnChar(nChar, nRepCnt, nFlags);
    }
}

else if (nChar == '.')
{
    for (int i = 0; i < text.GetLength(); i++)
    {
        if (text[i] == '.')
            return ;
    }

    for (i = 0; i < text.GetLength(); i++)
```



```

    {
        if (((text[i] == 'e') || (text[i] == 'E')) && (caret.x > i))
            return ;
    }

    CEdit::OnChar(nChar, nRepCnt, nFlags);
}

```

在上述代码中,如果用户的输入是数字键、E、-、退格键或 e,则调用 CEdit 类的默认处理函数;否则直接返回不作任何处理,而其结果就是按下其他键时,编辑控件不会有任何显示。

2. 编辑十六进制数值

这里将向读者介绍相对复杂一些的例子。使用过 Pctools 或 FPE 的读者,可能都会对其中的十六进制文件编辑器留有深刻的印象。正是使用这些工具,我们才能轻松地修改文件或游戏数据。下面就给出如何创建十六进制编辑器的方法,图 3-5 所示即为编辑器的运行界面。

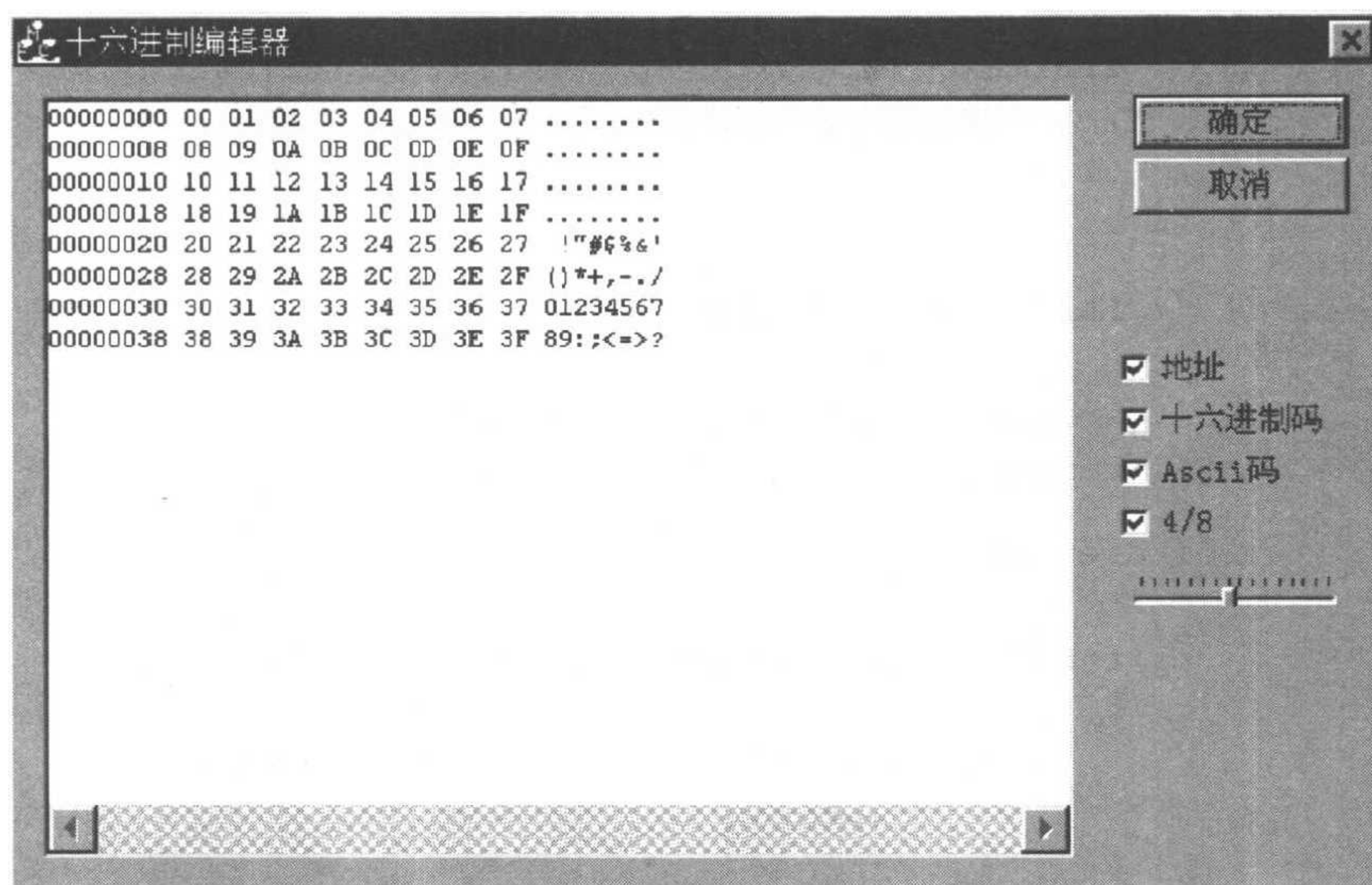


图 3-5 十六进制编辑器

首先创建编辑器的管理类 CHexEdit,与双精度数值输入处理类似,在 CHexEdit 类中也需要处理 WM_CHAR 消息,如清单 3-10 所示:

清单 3-10 OnChar()函数

```

void CHexEdit::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    if(! m_pData)
        return;
    if(nChar == '\\t')

```

```

        return;
    if(GetKeyState(VK_CONTROL) & 0x80000000)
    {
        switch(nChar)
        {
            case 0x03:
                if(IsSelected())
                    OnEditCopy();
                return;
            case 0x16:
                OnEditPaste();
                return;
            case 0x18:
                if(IsSelected())
                    OnEditCut();
                return;
            case 0x1a:
                OnEditUndo();
                return;
        }
    }

    if(nChar == 0x08)
    {
        if(m_currentAddress > 0)
        {
            m_currentAddress--;
            SelDelete(m_currentAddress, m_currentAddress + 1);
            RepositionCaret(m_currentAddress);
            RedrawWindow();
        }
        return;
    }

    SetSel(-1, -1);
    switch(m_currentMode)
    {
        case EDIT_NONE:
            return;
        case EDIT_HIGH:
        case EDIT_LOW:
            if((nChar >= '0' && nChar <= '9') || (nChar >= 'a' && nChar <= 'f'))
            {
                UINT b = nChar - '0';
                if(b > 9)
                    b = 10 + nChar - 'a';

                if(m_currentMode == EDIT_HIGH)
                {
                    m_pData[m_currentAddress] = (unsigned
                        char)((m_pData[m_currentAddress] & 0x0f) | (b << 4));
                }
            }
        }
    }

```

```

        }
        else
        {
            m_pData[m_currentAddress] = (unsigned
            char)((m_pData[m_currentAddress] & 0xf0)|b);
        }
        Move(1,0);
    }
    break;
case EDIT_ASCII:
    m_pData[m_currentAddress] = (unsigned char)nChar;
    Move(1,0);
    break;
}
RedrawWindow();
}

```

由于 CHexEdit 类管理的是一个编辑器,除了对十六进制数字键进行响应外,还必须具备编辑器的一些功能(例如拷贝、粘贴等)。因此,在 OnChar 函数中还需要对特殊组合键(例如 Ctrl + C、Ctrl + V)等进行处理。需要注意的是,这里的编辑函数需要定制,由于本书主要讨论界面制作,因此就不对这些函数再加讲解,请读者参考配套光盘 chap4/hexedit 目录下的源代码。

既然是编辑器,那么就需要对编辑时常用的箭头键、End、Home、Delete 等键作出响应。如果读者稍微留心一下十六进制编辑器的界面,就会发现其中的数据与空格是相互交错分布的,而光标在空格与数据区中的移动方式是不同的。因此,也不能使用 CEdit 类的默认处理,而应该自己定制。在前面的内容中已经提到过,当按下可显示字符时,会触发 WM_CHAR 消息;而如果按下的键为不可显示字符,则只会触发 WM_KEYDOWN 和 WM_KEYUP 消息。因此,在 CHexEdit 类中添加了对 WM_KEYDOWN 消息的处理,如清单 3-11 所示:

清单 3-11 OnKeyDown()函数

```

void CHexEdit::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    BOOL bShift = GetKeyState(VK_SHIFT) & 0x80000000;
    BOOL bac = m_bNoAddressChange;
    m_bNoAddressChange = TRUE;
    switch(nChar)
    {
        case VK_DOWN:
            if(bShift)
            {
                if(! IsSelected())
                {
                    m_selStart = m_currentAddress;
                }
                Move(0,1);
            }
        }
    }
}

```

```
m_selEnd = m_currentAddress;
if(m_currentMode == EDIT_HIGH || m_currentMode == EDIT_LOW)
    m_selEnd++;
RedrawWindow();
break;
}
else
    SetSel(-1, -1);
Move(0,1);
break;
case VK_UP:
    if(bShift)
    {
        if(! IsSelected())
        {
            m_selStart = m_currentAddress;
        }
        Move(0, -1);
        m_selEnd = m_currentAddress;
        RedrawWindow();
        break;
    }
    else
        SetSel(-1, -1);
    Move(0, -1);
    break;
case VK_LEFT:
    if(bShift)
    {
        if(! IsSelected())
        {
            m_selStart = m_currentAddress;
        }
        Move(-1,0);
        m_selEnd = m_currentAddress;
        RedrawWindow();
        break;
    }
    else
        SetSel(-1, -1);
    Move(-1,0);
    break;
case VK_RIGHT:
    if(bShift)
    {
        if(! IsSelected())
        {
            m_selStart = m_currentAddress;
```



```
        }
        Move(1,0);
        m_selEnd = m_currentAddress;
        if(m_currentMode == EDIT_HIGH || m_currentMode == EDIT_LOW)
            m_selEnd++;
        RedrawWindow();
        break;
    }
    else
        SetSel(-1, -1);
    Move(1,0);
    break;
case VK_PRIOR:
    if(bShift)
    {
        if(! IsSelected())
        {
            m_selStart = m_currentAddress;
        }
        OnVScroll(SB_PAGEUP, 0, NULL);
        Move(0,0);
        m_selEnd = m_currentAddress;
        RedrawWindow();
        break;
    }
    else
        SetSel(-1, -1);
    OnVScroll(SB_PAGEUP, 0, NULL);
    Move(0,0);
    break;
case VK_NEXT:
    if(bShift)
    {
        if(! IsSelected())
        {
            m_selStart = m_currentAddress;
        }
        OnVScroll(SB_PAGEDOWN, 0, NULL);
        Move(0,0);
        m_selEnd = m_currentAddress;
        RedrawWindow();
        break;
    }
    else
        SetSel(-1, -1);
    OnVScroll(SB_PAGEDOWN, 0, NULL);
    Move(0,0);
    break;
```

```
case VK_HOME:
    if(bShift)
    {
        if(! IsSelected())
        {
            m_selStart = m_currentAddress;
        }
        if(GetKeyState(VK_CONTROL) & 0x80000000)
        {
            OnVScroll(SB_THUMBTRACK, 0, NULL);
            Move(0,0);
        }
        else
        {
            m_currentAddress /= m_bpr;
            m_currentAddress * = m_bpr;
            Move(0,0);
        }
        m_selEnd = m_currentAddress;
        RedrawWindow();
        break;
    }
    else
        SetSel(-1, -1);
    if(GetKeyState(VK_CONTROL) & 0x80000000)
    {
        OnVScroll(SB_THUMBTRACK, 0, NULL);
        m_currentAddress = 0;
        Move(0,0);
    }
    else
    {
        m_currentAddress /= m_bpr;
        m_currentAddress * = m_bpr;
        Move(0,0);
    }
    break;
case VK_END:
    if(bShift)
    {
        if(! IsSelected())
        {
            m_selStart = m_currentAddress;
        }
        if(GetKeyState(VK_CONTROL) & 0x80000000)
        {
            m_currentAddress = m_length-1;
            OnVScroll(SB_THUMBTRACK, ((m_length + (m_bpr/2)) /
            m_bpr) - m_lpp,
```

```

        NULL);
        Move(0,0);
    }
    else
    {
        m_currentAddress /= m_bpr;
        m_currentAddress * = m_bpr;
        m_currentAddress += m_bpr - 1;
        if(m_currentAddress > m_length)
            m_currentAddress = m_length-1;
        Move(0,0);
    }
    m_selEnd = m_currentAddress;
    RedrawWindow();
    break;
}
else
    SetSel(-1, -1);
if(GetKeyState(VK_CONTROL) & 0x80000000)
{
    m_currentAddress = m_length-1;
    if(m_bHalfPage)
        OnVScroll(SB_THUMBTRACK, 0, NULL);
    else
        OnVScroll(SB_THUMBTRACK, ((m_length + (m_bpr/2)) /
            m_bpr) - m_lpp,
            NULL);
    Move(0,0);
}
else
{
    m_currentAddress /= m_bpr;
    m_currentAddress * = m_bpr;
    m_currentAddress += m_bpr - 1;
    if(m_currentAddress > m_length)
        m_currentAddress = m_length-1;
    Move(0,0);
}
break;
case VK_INSERT:
    SelInsert(m_currentAddress, max(1, m_selEnd-m_selStart));
    RedrawWindow();
    break;
case VK_DELETE:
    if(IsSelected())
    {
        OnEditClear();
    }
    else

```

```

        {
            SelDelete(m_currentAddress, m_currentAddress + 1);
            RedrawWindow();
        }
        break;
    case '\t':
        switch(m_currentMode)
        {
            case EDIT_NONE:
                m_currentMode = EDIT_HIGH;
                break;
            case EDIT_HIGH:
            case EDIT_LOW:
                m_currentMode = EDIT_ASCII;
                break;
            case EDIT_ASCII:
                m_currentMode = EDIT_HIGH;
                break;
        }
        Move(0,0);
        break;
    }
    m_bNoAddressChange = bac;
}

```

3.4.2 在位编辑

在使用列表框、组合框、列表视图等控件时,经常需要对其中的条目文本加以修改。常规的方法是在控件边添加一个“修改”按钮。以列表框为例,当用户单击该按钮时,就会弹出一个对话框,供用户在其中修改被选中的条目文本。不过这显然不是最有效的方法。更好的选择应该是在将修改的字符串上方,动态创建并显示编辑框(一般应该在双击某条目后进行)。而且这个动态创建的控件还可以被重复使用。下面我们就将根据这一思路,创建在位编辑控件,并将其管理类命名为 CSmartEdit。

首先应该确定一个修改完成标志,这里将其设置为 Enter 键。也就是说,当用户按下 Enter 键时,在位编辑控件就向其父窗口发送通告。不过父窗口(例如拥有列表框的对话框)怎么才能得到用户输入的新字符串呢?一个方法就是在创建 CSmartEdit 对象时,将 this 指针传递给它。这样对话框就会在接到结束通告后,立即调用对话框函数来接收新的字符串。但是这种方法有两个缺点:其一需要知道父窗口的具体类型;其二不同的父窗口需要实现同一个函数以接收新字符串。

一个解决方法可以是这样:向父窗口发送一个消息,但其功能仅仅是告诉父窗口“用户可能已经修改了字符串”;接着 CSmartEdit 对象将其中包含的文本拷贝到剪贴板中。这样,CSmartEdit 就无需关心父窗口的类型,而父窗口只要使用剪贴板中的数据即可。

假设双击列表框的条目,就表示开始编辑,那么就应该在对话框中响应 LBN_

DBLCLK 消息(列表框中的某条目双击),函数的示范实现如清单 3-12 所示:

清单 3-12 OnDblclkStringInListBox() 函数

```
void myDialog::OnDblclkStringInListBox()
{
    const INT nIndex = m_Ctl.GetCurSel();
    if(nIndex == LB_ERR) return;
    CString string;
    m_Ctl.GetText(nIndex, string);
    RECT rect;
    INT result = m_Ctl.GetItemRect(nIndex, &rect);
    if(result == LB_ERR) return;
    CSmartEdit * pEdit = new SmartEdit;
    rect.bottom += 4;
    pEdit -> Create(WS_CHILD | WS_VISIBLE | WS_BORDER | ES_AUTOHSCROLL,
        rect, &m_Ctl, (UINT)-1);
    pEdit -> SetWindowText(string);
    pEdit -> SetFocus();
    pEdit -> LimitText(MYMAX_LEN);
}
```

在上面的代码中,首先确定是否有条目被选中,然后将获得被编辑的字符串。接着取得条目的边界矩形,并使用该矩形确定在位编辑框的尺寸和位置。

清单 3-13 所示为 CSmartEdit 类的实现代码:

清单 3-13 CSmartEdit 类的实现代码

```
CSmartEdit::CSmartEdit() : bEscapeKey(FALSE)
{
}

BEGIN_MESSAGE_MAP(SmartEdit, CEdit)
//{{AFX_MSG_MAP(SmartEdit)
ON_WM_KILLFOCUS()
//{{AFX_MSG_MAP
END_MESSAGE_MAP()

void CSmartEdit::OnKillFocus(CWnd *)
{
    PostMessage(WM_CLOSE, 0, 0);
    if(! bEscapeKey){
        CString str;
        GetWindowText(str);
        COleDataSource * pds = new COleDataSource;
        PTCHAR cp = (PTCHAR)GlobalAlloc(GMEM_FIXED, (str.GetLength() *
            sizeof(TCHAR)) + sizeof(TCHAR));
        _tcscpy(cp, str);
        pds -> CacheGlobalData(CF_TEXT, cp);
        pds -> SetClipboard();
        GetOwner() -> PostMessage(EDITCLASSMSG);
    }
}
```

```

    }
}

void CSmartEdit::PostNcDestroy()
{
    delete this;
}

BOOL CSmartEdit::PreTranslateMessage(MSG * pMsg)
{
    if(pMsg->wParam == VK_RETURN){
        PostMessage(WM_CLOSE, 0, 0);
        return TRUE;
    }else if(pMsg->wParam == VK_ESCAPE){
        PostMessage(WM_CLOSE, 0, 0);
        return bEscapeKey = TRUE;
    }

    return CEdit::PreTranslateMessage(pMsg);
}

```

由于 CSmartEdit 对象是在对话框对象 myDialog 中使用 new 运算符创建的。因此必须在控件被销毁时使用 delete 将其删除,以免出现内存泄漏。而 PostNcDestroy 函数是在窗口被销毁后被调用,因此这是删除对象的最好位置。

类的主要功能是在 OnKillFocus 函数中实现的。当 CSmartEdit 失去输入焦点时(不是由于按下 Escape 键),就调用 GetWindowText 函数以得到其中的新字符串。接着将该字符串通告 COleDataSource 对象放置到剪贴板中。需要注意的是,使用 PTCHAR 和 _tcsncpy 函数的主要目的是使控件同时适用于 ANSI 和 UNICODE 应用程序。当字符串被拷贝到剪贴板中后,跟着就发送 EDITCLASSMSG 消息给其父窗口。EDITCLASSMSG 是自定义的消息,例如将其定义为 WM_APP + 100。

既然 CSmartEdit 向父窗口发送了 EDITCLASSMSG 消息,那么父窗口就需要处理这个消息。清单 3-14 所示为 PreTranslateMessage 函数的源代码:

清单 3-14 PreTranslateMessage() 函数

```

BOOL myDialog::PreTranslateMessage(MSG * pMsg)
{
    if(EDITCLASSMSG == pMsg->message){
        COleDataSource data;

        hClipboard()){
            if(data.IsDataAvailable(CF_TEXT)){
                HGLOBAL hg;
                if(hg = data.GetGlobalData(CF_TEXT)){
                    CString str = (LPCTSTR)GlobalLock(hg); GlobalUnlock
                    (hg);
                    NewString(str); // 应用程序定义的处理新字符串的函数
                }
            }
        }
    }
}

```

```

        data.Release();
    }
    return TRUE;
}

return CDialog::PreTranslateMessage(pMsg);
}

```

当父窗口接收到 EDITCLASSMSG 消息后,就等于得到了剪贴板中有新文本的通知。因此在上述代码中使用 COleDataObject 对象将其取回。而被取回的文本则由 NewString 函数完成设置。当然,在应用程序中还应该保存被编辑条目的索引以便修改。而这应该在 OnDblclkStringInListBox 函数中完成。例如,使用“const INT nIndex = m_Ctl.GetCurSel();”代替原来的“m_nIndex = m_Ctl.GetCurSel();”。在 NewString 函数中则使用 myDialog 的成员变量 m_nIndex 来定位将被修改的新字符串。

现在已经完成了在位编辑所需的全部步骤。不过可能读者还会有个疑问:既然在 PostNcDestroy 函数中 CSmartEdit 对象已经被删除,那么它就必须是使用 new 在堆上创建的。我们无法阻止用户试图使用 DDX_Control 来完成数据的交换,而这会在调用 delete 时导致程序崩溃。因此,必须强制使用 new 运算符。那么应该怎么做呢?答案很简单,只要将类的析构函数 ~CSmartEdit 声明为 protected 即可。这将禁止类对象在堆栈上被创建。

3.4.3 语法着色

所谓语法着色,就是将关键字以不同的颜色显示,这通常用在代码编辑器中,Develop Studio 就是一个很好的例子。

实现语法着色的思路并不复杂:首先创建一个关键字和颜色的映射,然后根据输入的单词决定其颜色。读者也许会有疑问,在 3.1 节中不是说编辑控件只能使用一种字体吗?那么又是怎么实现不同颜色的文本输出呢,实际上这是绘制完成的。虽然 CEdit 控件不支持使用多种字体,但使用不同的颜色绘制文本却是完全可以的。清单 3-15 所示为 DrawSingleLine 的部分源代码,读者可以清楚地看到其绘制过程:

清单 3-15 DrawSingleLine() 函数

```

void CCrystalTextView::DrawSingleLine(CDC * pdc, const CRect &rc, int nIndex)
{
    ASSERT(nLineIndex >= -1 && nIndex < GetLineCount());

    if (nLineIndex == -1)
    {
        // 绘制文本上的线
        pdc->FillSolidRect(rc, GetColor(COLORINDEX_WHITE_SPACE));
        return;
    }
}

```

```

// 得到当前行的背景颜色
BOOL bDrawWhitespace = FALSE;
COLORREF crBkgnd, crText;
GetLineColors(nLineIndex, crBkgnd, crText, bDrawWhitespace);
if (crBkgnd == CLR_NONE)
    crBkgnd = GetColor(COLORINDEX_BKGND);

int nLength = GetLineLength(nLineIndex);
if (nLength == 0)
{
    // 绘制空行
    CRect rect = rc;
    if ((m_bFocused || m_bShowInactiveSelection) && IsInsideSelBlock(
        CPoint(0, nLineIndex)))
    {
        pdc->FillSolidRect(rect.left, rect.top, GetCharWidth(), rect.
            Height(),
            GetColor(COLORINDEX_SELBKGND));
        rect.left += GetCharWidth();
    }
    pdc->FillSolidRect(rect, bDrawWhitespace ? crBkgnd :
        GetColor(COLORINDEX_WHITESPACE));
    return;
}

// 分析行
LPCTSTR pszChars = GetLineChars(nLineIndex);
DWORD dwCookie = GetParseCookie(nLineIndex - 1);
TEXTBLOCK * pBuf = (TEXTBLOCK *)_alloca(sizeof(TEXTBLOCK) * nLength * 3);
int nBlocks = 0;
m_pdwParseCookies[nLineIndex] = ParseLine(dwCookie, nLineIndex, pBuf,
    nBlocks);
ASSERT(m_pdwParseCookies[nLineIndex] != (DWORD) - 1);

// 绘制文本
CPoint origin(rc.left - m_nOffsetChar * GetCharWidth(), rc.top);
pdc->SetBkColor(crBkgnd);
if (crText != CLR_NONE)
    pdc->SetTextColor(crText);
BOOL bColorSet = FALSE;

...
}

```

上面讲述了语法着色的基本思路,在具体实现时还有一些需要注意的细节,但是这已

不在本书的讨论范围之内了,请读者自行参考配套光盘的 chap3/syntaxcolor 目录下的源代码。

本章小结

本章主要向读者介绍了如何修改常规 Windows 编辑控件,通过本章学习读者应该达到以下几点:

- 掌握 CEdit 和 CEditView 类的使用。
- 掌握定制编辑控件的方法。

第 4 章 组合框控件

在前面两章中介绍的按钮控件和编辑控件的实际使用相对较为简单,从本章开始将介绍一些使用较为复杂的控件。本章介绍组合框控件,该控件主要用于为用户提供选项编辑和选择功能,是 Windows 软件中的最常用控件之一。

本章要点:

- CComboBox 类的使用;
- 改变组合框控件的行为;
- 改变组合框控件中的选项形式。

4.1 组合框控件编程基础

组合框由列表框与静态控件或编辑控件共同组成。列表框部分用于显示选项,或当用户选择下拉箭头时,可以下拉显示选项。CComboBox 类为 Windows 组合框控件的实现提供了功能支持,其派生结构如图 4-1 所示。

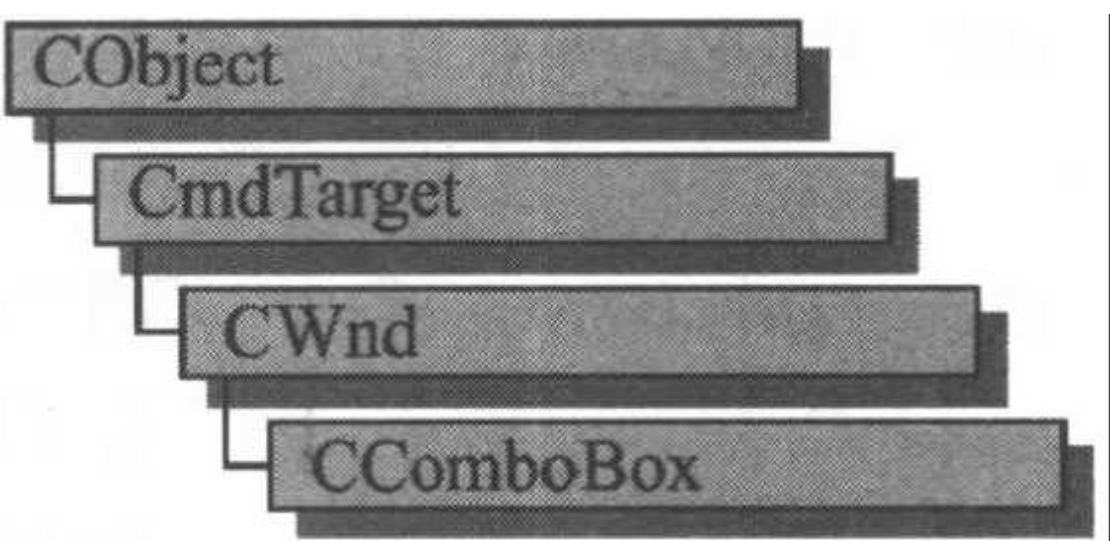


图 4-1 CComboBox 类的派生结构

4.1.1 组合框控件概述

如果用户选中列表框中的某项,则该项将显示在静态或编辑控件中。此外,如果组合框中的列表框为下拉风格,则用户可以在编辑控件中输入某项,此时如果列表框可见,其中的对应项将被高亮显示。表 4-1 比较了组合框的不同风格。

表 4-1 组合框的风格比较

组合框风格	列表框	静态/编辑控件
简单	总是可见	编辑控件
下拉式	下拉时可见	编辑控件
下拉列表式	下拉时可见	静态控件

用户既可以在对话框模板中创建组合框控件,也可以在代码中直接创建。在这两种情况下,都需要首先调用构造函数 CComboBox 以创建 CComboBox 对象,接着调用 Create 成员函数以创建 Windows 组合框控件,并将其与 CComboBox 对象连接。

如果希望处理由按钮控件向其父窗口(通常是对话框)发送的 Windows 通告消息,则

需要在相应的父窗口类中添加消息映射入口和消息处理函数。

每个消息映射入口都具有以下形式：

```
ON_Notification(id, memberFxn)
```

其中 id 指定了发送通告消息的控件 ID, 而 memberFxn 则指定了用于处理控件通告消息的成员函数。

而消息处理函数如下：

```
afx_msg void memberFxn( );
```

通告向父窗口发送的顺序是不可预测的。例如, CBN_SELCHANGE 通告可能在 CBN_CLOSEUP 通告前发送, 也可能在其后才发送。组合框控件常用的通告如表 4-2 所示。

表 4-2 组合框控件通告

通告	含义
CBN_CLOSEUP (适用于 Windows 3.1 或以后版本)	组合框中的列表框被关闭。具有 CBS_SIMPLE 风格的组合框(简单组合框)不会发送该通告
CBN_DBLCLK	用户双击组合框的列表框中的某项。只有具有 CBS_SIMPLE 风格的组合框才会发送该通告。在具有 CBS_DROPDOWN 或 CBS_DROPDOWNLIST 风格的组合框中不会发生双击操作, 因为单击就会使列表框隐藏
CBN_DROPDOWN	组合框的列表框将被下拉(可见)。只有具有 CBS_DROPDOWN 或 CBS_DROPDOWNLIST 风格的组合框会发送该通告
CBN_EDITCHANGE	用户改变了组合框的编辑框中的内容。该通告与 CBN_EDITUPDATE 通告不同, 它在 Windows 更新屏幕后才发送。具有 CBS_DROPDOWNLIST 风格的组合框不能发送该通告
CBN_EDITUPDATE	组合框的编辑框部分将显示改变后的文本。该通告在控件将文本格式化后, 但更新显示前发送。具有 CBS_DROPDOWNLIST 风格的组合框不能发送该通告
CBN_ERRSPACE	组合框不能为请求分配足够的内存
CBN_SELENDCANCEL (适用于 Windows 3.1 或以后版本)	该通告表示用户选择将被取消, 它在 CBN_CLOSEUP 通告前发送。一般来说当用户选择了一个选项后, 接着单击其他窗口或控件, 则会发送该通告。不过即使没有发送 CBN_CLOSEUP 通告, CBN_SELENDCANCEL 或 CBN_SELENDOK 通告也会被发送
CBN_SELENDOK	用户选择了某个选项后, 接着单击 Enter 键或单击下拉按钮时会发送该通告, 它在 CBN_CLOSEUP 通告前发送。
CBN_KILLFOCUS	组框失去输入焦点
CBN_SELCHANGE	用户改变了组合框中的选项。当发送该通告时, 如果需要得到编辑控件中的文本, 则只能通过类似 GetLBText 的函数获得, 而不能使用 GetWindowText 函数
CBN_SETFOCUS	组合框得到输入焦点

如果在对话框中创建组合框,则组合框对象将在对话框关闭时自动销毁。如果在其他的 Windows 对象中使用嵌入组合框对象,也无需手动销毁它。只有当使用 new 函数在堆上创建组合框对象时,才需要在使用完毕后,调用 delete 函数销毁它。

4.1.2 构造函数

CComboBox 类的构造函数包括: CComboBox、Create 和 InitStorage,它们可以完成构造 CButton 对象,创建 Windows 组合框控件等功能。

- CComboBox

调用该函数以创建一个 CComboBox 对象,其原型为:

```
CComboBox( );
```

- Create

调用该函数以创建组合框并将其与 CComboBox 对象相联系,其原型为:

```
BOOL Create( DWORD dwStyle, const RECT& rect, CWnd * pParentWnd, UINT nID );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

dwStyle —— 指定了组合框的风格,其取值如表 4-3 所示。

表 4-3 dwStyle 参数取值

dwStyle 参数取值	含义
CBS_AUTOHSCROLL	当用户在编辑框中文本的末端进行输入时,自动滚动到文本最右端。如果不设置该风格,文本长度不能超过编辑框矩形的范围
CBS_DROPDOWN	与 CBS_SIMPLE 风格类似,只是当用户单击下拉按钮时,才显示列表框
CBS_DROPDOWNLIST	与 CBS_DROPDOWN 风格类似,只是使用静态控件而不是编辑控件来显示当前列表框中的选项
CBS_HASSTRINGS	组合框为自绘制控件,且其中选项由字符串组成。组合框负责维护字符串的指针和内存,这样应用程序可以使用 GetText 成员函数得到指定选项的文本
CBS_OEMCONVERT	将在组合框的编辑控件中输入的文本,从 ANSI 字符集转换为 OEM 字符集,然后再将其转换回 ANSI 字符集。这样,确保了 AnsiToOem 函数将组合框中的 ANSI 字符串转换为 OEM 字符串操作的正确性。该风格在组合框中包含文件名时尤其有用。只有具有 CBS_SIMPLE 或 CBS_DROPDOWN 的组合框能够使用该风格
CBS_OWNERDRAWFIXED	组合框的父窗口负责绘制组合框中的内容,并且列表框中的所有选项都具有同样的高度

续表

dwStyle 参数取值	含义
CBS_OWNERDRAWVARIABLE	组合框的父窗口负责绘制组合框中的内容,并且列表框中的选项高度不同
CBS_SIMPLE	列表框中显示所有选项,当前选项显示在编辑框中
CBS_SORT	自动排列列表框中的选项
CBS_DISABLENOSCROLL	当列表框足以容纳所有选项时,禁止右边的垂直滑块。如果不指定该风格,则当列表框足以容纳所有选项时,隐藏垂直滑块
CBS_NOINTEGRALHEIGHT	组合框的尺寸必须为在代码中给出的尺寸。通常,Windows 会调整组合框的尺寸,以使其显示所有选项

rect —— 指定了组合框的尺寸和位置,该参数可以为 RECT 结构或 CRect 对象。

pParentWnd —— 指定了组合框的父窗口,通常为对话框。该参数不能为 NULL。

nID —— 指定了列表框控件的 ID。

创建 CComboBox 对象需要两个步骤:首先调用构造函数,然后调用 Create 来创建 Windows 组合框,并将其与 CComboBox 对象相连接。

当执行 Create 函数时,Windows 将 WM_NCCREATE、WM_CREATE、WM_NCCALCSIZE 和 WM_GETMINMAXINFO 消息发送给组合框。默认情况下,这些消息由 CWnd 类的 OnNcCreate、OnCreate、OnNcCalcSize 和 OnGetMinMaxInfo 成员函数处理。如果希望增强对组合框控件的控制,用户可以在代码中重载这些函数。例如,在创建窗口时常常重载 OnCreate 函数以添加定制的初始操作。

能够对组合框控件使用的窗口风格常数如表 4-4 所示。

表 4-4 能对组合框控件使用的窗口风格常数

风格常数	含义
WS_CHILD	子窗口
WS_VISIBLE	窗口控件
WS_DISABLED	窗口被禁止
WS_VSCROLL	添加垂直滚动滑块
WS_HSCROLL	添加水平滚动滑块
WS_GROUP	成组按钮
WS_TABSTOP	使按钮能够用 Tab 键切换

• InitStorage

调用该函数以为列表框中选项和字符串分配内存块,其原型为:

```
int InitStorage( int nItems, UINT nBytes );
```

返回值:

如果函数调用成功,则返回需要重新分配内存前,在列表框中可容纳的最大选项数目;否则,返回 CB_ERR,表示没有足够的可用内存。

参数:

nItems —— 指定了将添加的选项数目。

nBytes —— 指定了要为选项字符串分配的内存字节数。

在向组合框的列表框中添加大量选项前,应该调用该函数预先分配内存。对于高于 Windows 95 的 Windows 版本, wParam 参数为 16 位值。这意味着,列表框中最大只能容纳 32,767 个选项。虽然对选项的数目有所限制,但实际上真正的限制来自可用的内存量。

对于包含大量选项(大于 100)的组合框, InitStorage 函数能够加速列表框的初始化。它预先分配了指定量的内存,这样接下来的 AddString、InsertString 和 Dir 函数耗费较少的时间。如果分配的内存量不足,则对于过多的选项将使用常规内存分配;如果分配的内存量过多,则会降低系统效率。

4.1.3 常规操作函数

CComboBox 类的常规操作函数包括 GetCount、GetCurSel、SetCurSel、GetEditSel、SetEditSel、SetItemData、SetItemDataPtr、GetItemDataPtr、GetItemData、GetItemDataPtr、GetTopIndex、SetTopIndex、SetHorizontalExtent、GetHorizontalExtent、SetDroppedWidth、GetDroppedWidth、Clear、Copy、Cut、Paste、LimitText、SetItemHeight、GetItemHeight、GetLBText、GetLBTextLen、ShowDropDown、GetDroppedControlRect、GetDroppedState、SetExtendedUI、GetExtendedUI、GetLocal 和 SetLocal,它们能够完成选项数目的检索、选项的设置等操作。

- GetCount

调用该函数以得到组合框中的选项数目,其原型为:

```
int GetCount( ) const;
```

返回值:

如果函数调用成功,则返回选项数目。返回值比最后一个选项的索引值大 1(索引以 0 开始计数)。如果函数调用失败,则返回 CB_ERR。

- GetCurSel

调用该函数以得到当前选项的索引,其原型为:

```
int GetCurSel( ) const;
```

返回值:

如果函数调用成功,则返回当前选项的索引。如果函数调用失败,则返回 CB_ERR。

- SetCurSel

调用该函数以设置组合框中的当前选项,其原型为:

```
int SetCurSel( int nSelect );
```

返回值:

如果函数调用成功,则返回当前选项的索引。如果 nSelect 的值超出范围,或为 -1,则返回 CB_ERR,并清除当前选项。

参数:

nSelect —— 指定了将被选择的选项的索引。如果该参数为 -1, 则清除当前组合框中的选择。

执行 SetCurSel 函数时, 如果必要会滚动列表框(如果列表框可见), 并且编辑控件中的文本也发生相应改变, 以反映在列表框中的选项变化。

- GetEditSel

调用该函数以得到在组合框的编辑框中当前被选文本的起始和终止位置, 其原型为:

```
DWORD GetEditSel( ) const;
```

返回值:

如果函数调用成功则返回一个 32 位值, 其中低位字为被选文本的起始位置, 而高位字为被选文本后的第一个未选字符。如果对没有编辑控件的组合框调用该函数, 则返回 CB_ERR。

- SetEditSel

调用该函数以选择组合框的编辑框中的字符, 其原型为:

```
BOOL SetEditSel( int nStartChar, int nEndChar );
```

返回值:

如果函数调用成功, 则返回非零值。如果组合框没有列表框或为 CBS_DROPDOWNLIST 风格, 则返回 CB_ERR。

参数:

nStartChar —— 指定了选择文本的起始位置。如果该参数为 -1, 则将去除任何当前的选项。

nEndChar —— 指定了选择文本的终止位置。如果该参数为 -1, 则将选择从起始位置到文本末端的所有字符。

函数使用的位置都是从 0 开始计算的, 也就是说, 如果要选择编辑控件中的第一个字符, 就需要将 nStartChar 设置为 0。而终止位置则为要选文本后的第一个字符, 例如, 如果希望选择编辑控件中的前 4 个字符, 则应该将起始位置和终止位置分别设置为 0 和 4。

- SetItemData

调用该函数以设置与组合框中选项有关的 32 位值, 其原型为:

```
int SetItemData( int nIndex, DWORD dwItemData );
```

返回值:

如果函数调用成功, 则返回值非 CB_ERR, 否则返回 CB_ERR。

参数:

nIndex —— 指定了将设置的选项的索引。

dwItemData —— 指定了将设置的新值。

- SetItemDataPtr

调用该函数以使用 void 指针设置选项的 32 位值, 其原型为:

```
int SetItemDataPtr( int nIndex, void* pData );
```

返回值:

如果函数调用成功,则返回值非 CB_ERR, 否则返回 CB_ERR。

参数:

nIndex —— 指定了将获取其值的选项索引。

pData —— 指定了将用以设置选项值的 void 指针。

函数中使用的指针将在组合框的生命期内保持有效,即使对应选项的相对位置发生改变(由于组合框中选项的增加和删除)。也就是说,选项的索引可以改变,而其指针却保持不变。

- GetItemData

调用该函数以得到组合框中指定选项的 32 位值,其原型为:

```
DWORD GetItemData( int nIndex ) const;
```

返回值:

如果函数调用成功,则返回选项的 32 位值,否则返回 CB_ERR。

参数:

nIndex —— 指定了组合框中的选项索引。

- GetItemDataPtr

调用该函数以 void 指针的形式得到组合框中指定选项的 32 位值,其原型为:

```
void* GetItemDataPtr( int nIndex ) const;
```

返回值:

如果函数调用成功,则返回 void 指针,否则返回 -1。

参数:

nIndex —— 指定了组合框中选项的索引。

- GetTopIndex

调用该函数以得到组合框中第一个可见选项的索引,其原型为:

```
int GetTopIndex( ) const;
```

返回值:

如果函数调用成功,则返回组合框中第一个可见选项的索引,否则返回 CB_ERR。在初始状态下,第一个可见选项的索引为 0,不过如果滚动列表框后,第一个可见索引的值就会发生变化。

- SetTopIndex

调用该函数以设置组合框中的第一个可见选项,其原型为:

```
int SetTopIndex( int nIndex );
```

返回值:

如果函数调用成功,则返回零值,否则返回 LB_ERR。

参数:

nIndex —— 指定了将被设置为第一个可见选项的索引值。

SetTopIndex 函数能够确保特定的选项在组合框的列表框中可见,此时系统将负责滚动列表框直到指定选项出现,或到达滚动的最大范围。

- SetHorizontalExtent

调用该函数以设置组合框中列表框部分的水平滚动范围(以像素为单位),其原型为:

```
void SetHorizontalExtent( UINT nExtent );
```

参数:

nExtent —— 指定了组合框中列表框的水平滚动范围。

SetHorizontalExtent 函数设置组合框中列表框的水平滚动范围。如果列表框的宽度小于该范围,则可以使用水平滑块滚动查看选项。如果列表框的宽度大于或等于该范围,则水平滑块被隐藏。

- GetHorizontalExtent

调用该函数以得到组合框中列表框部分的水平滚动范围,其原型为:

```
UINT GetHorizontalExtent( ) const;
```

返回值:

如果函数调用成功,则返回组合框中列表框部分的水平滚动范围。

只有当组合框具有水平滑块时,该函数才有用。

- SetDroppedWidth

调用该函数以设置组合框的下拉式列表框的最小宽度(以像素为单位),其原型为:

```
int SetDroppedWidth( UINT nWidth );
```

返回值:

如果函数调用成功,则返回设置后列表框的新宽度,否则返回 CB_ERR。

参数:

nWidth —— 指定了将为组合框设置的列表框宽度。

SetDroppedWidth 函数只能用于具有 CBS_DROPDOWN 或 CBS_DROPDOWNLIST 风格的组合框。在默认情况下,下拉式列表框的最小允许宽度为 0。当显示组合框的列表框部分时,其宽度将大于最小允许宽度或组合框宽度。

- GetDroppedWidth

调用该函数以得到组合框的下拉式列表框的最小宽度(以像素为单位),其原型为:

```
int GetDroppedWidth( ) const;
```

返回值:

如果函数调用成功,则返回列表框的最小允许宽度,否则返回 CB_ERR。

GetDroppedWidth 函数只能用于具有 CBS_DROPDOWN 或 CBS_DROPDOWNLIST 风格的组合框。

- Clear

调用该函数以清除组合框编辑控件中显示的当前选项,其原型为:

```
void Clear( );
```

- Copy

调用该函数将编辑控件中的当前选项,以 CF_TEXT 格式拷贝到剪贴板中,其原型为:

```
void Copy( );
```

- Cut

调用该函数以清除组合框编辑控件中显示的当前选项,并将其以 CF_TEXT 格式拷贝到剪贴板中,其原型为:

```
void Cut();
```

- Paste

调用该函数以将剪贴板中的数据插入组合框编辑控件的光标处(只有当剪贴板中的数据为 CF_TEXT 格式时,才能完成该操作),其原型为:

```
void Paste();
```

- LimitText

调用该函数以用于向组合框编辑控件中输入的文本长度,其原型为:

```
BOOL LimitText( int nMaxChars );
```

返回值:

如果函数调用成功,则返回非零值。如果对没有编辑控件的组合框,或具有 CBS_DROPDOWNLIST 风格的组合框调用该函数,则返回 CB_ERR。

参数:

nMaxChars —— 指定了用户可以输入的文本长度。如果该参数为 0,则将输入文本的最大长度设置为 65,535 字节。

LimitText 函数只对用户输入有影响,而对在发送消息时已经存在于编辑控件中的文本,或已经存在的组合框选项无作用。也就是说,如果组合框中预设了一个长度大于 LimitText 函数所设置的长度的选项,那么当用户选择该选项时,编辑控件中将完全显示它,并不会因为 LimitText 函数的设置而有所改变。

- SetItemHeight

调用该函数以设置组合框中组元的高度,其原型为:

```
int SetItemHeight( int nIndex, UINT cyItemHeight );
```

返回值:

如果函数调用成功,则返回零值,否则返回 CB_ERR。

参数:

nIndex —— 指定了将设置的组合框组元。如果组合框具有 CBS_OWNERDRAWVAR-

TABLE 风格,则 nIndex 为将设置的组合框选项的索引;否则, nIndex 必须为零,并设置所有选项的高度。如果 nIndex 为 -1,则将设置组合框中的编辑控件或静态控件的高度。

cyItemHeight —— 指定了将设置的组合框组元高度,以像素为单位。

组合框的编辑控件或静态控件的高度,与其中选项的高度无关。因此应用程序必须确保它们的高度大于选项的高度,以便能够完全显示。

- **GetItemHeight**

调用该函数以获得组合框组元的高度,其原型为:

```
int GetItemHeight( int nIndex ) const;
```

返回值:

如果函数调用成功,则返回组合框中指定组元的像素高度,否则返回 CB_ERR。

参数:

nIndex —— 指定了将获取其高度的组合框组元。如果 nIndex 为 -1,则将获取组合框中的编辑控件或静态控件的高度。如果组合框具有 CBS_OWNERDRAWVARIABLE 风格,则 nIndex 为将获取其高度的组合框选项的索引;否则, nIndex 必须为零。

- **GetLBText**

调用该函数以获得组合框中指定选项文本,其原型为:

```
int GetLBText( int nIndex, LPTSTR lpszText ) const;  
void GetLBText( int nIndex, CString& rString ) const;
```

返回值:

如果函数调用成功,则返回字符串的字节长度(不包括末尾的空字符)。如果 nIndex 并非合法的索引,则返回 CB_ERR。

参数:

nIndex —— 指定了将获取其文本的选项索引。

lpszText —— 指定了将用以存放所得文本的缓冲区指针。该缓冲区必须足够容纳要获取的文本及其结尾空字符。

rString —— 指定了用以存放所得文本的 CString 对象。

- **GetLBTextLen**

调用该函数以得到组合框选项的长度,其原型为:

```
int GetLBTextLen( int nIndex ) const;
```

返回值:

如果函数调用成功,则返回字符串的字节长度(不包括末尾的空字符)。如果 nIndex 并非合法的索引,则返回 CB_ERR。

参数:

nIndex —— 指定了将获取的组合框选项索引。

- **ShowDropDown**

调用该函数以显示或隐藏组合框的列表框部分(组合框必须具有 CBS_DROPDOWN

或 CBS_DROPDOWNLIST 风格),其原型为:

```
void ShowDropDown( BOOL bShowIt = TRUE );
```

参数:

bShowIt —— 指定了列表框将被显示或隐藏。如果该参数为 TRUE,则显示列表框,否则隐藏列表框。

该成员函数对 CBS_SIMPLE 型组合框无效。

- GetDroppedControlRect

调用该函数以得到下拉式组合框的列表框的可视屏幕坐标,其原型为:

```
void GetDroppedControlRect( LPRECT lprect ) const;
```

参数:

lprect —— 为指向 RECT 结构的指针,用以存放返回的坐标。

- GetDroppedState

调用该函数以确定下拉式组合框的列表框是否可见,其原型为:

```
BOOL GetDroppedState( ) const;
```

返回值:

如果列表框可见,则返回非零值,否则返回零值。

- SetExtendedUI

调用该函数以决定对具有 CBS_DROPDOWN 或 CBS_DROPDOWNLIST 风格的列表框使用默认用户接口还是扩展用户接口,其原型为:

```
int SetExtendedUI( BOOL bExtended = TRUE );
```

返回值:

如果函数调用成功,则返回 CB_OKAY,否则返回 CB_ERR。

参数:

bExtended —— 指定了将对组合框使用的用户接口。如果该参数为 TRUE,则使用扩展用户接口,否则使用默认(标准)用户接口。

扩展用户接口有如下特征:

- (1) 对具有 CBS_DROPDOWNLIST 风格的列表框,单击其静态控件能够显示列表框。
- (2) 单击下拉按钮以显示列表框(禁止 F4)。
- (3) 当列表框被禁止时(下拉按钮被禁止),静态控件禁止滚动。

- GetExtendedUI

调用该函数以确定组合框使用的用户接口风格,其原型为:

```
BOOL GetExtendedUI( ) const;
```

返回值:

如果组合框使用的是扩展用户接口,则返回非零值,否则返回零值。

- GetLocale

调用该函数以得到组合框的本地标识符,其原型为:

```
LCID GetLocale( ) const;
```

返回值:

如果函数调用成功,则返回组合框中字符串的本地标识符(LCID)值。

- SetLocale

调用该函数以设置组合框的本地标识符,其原型为:

```
LCID SetLocale( LCID nNewLocale );
```

返回值:

如果函数调用成功,则返回先前设置的组合框的本地标识符。

参数:

nNewLocale —— 指定了将为组合框设置的新本地标识符值。

如果不调用 SetLocale 函数,则将使用由系统获得的本地标识符,而系统默认的本地标识符可以在控制面板中进行修改。

4.1.4 字符串操作函数

CComboBox 类的字符串操作函数包括: AddString、DeleteString、InsertString、ResetContent、Dir、FindString、FindStringExact 和 SelectString 函数,它们能够完成向组合框中添加、删除以及插入选项等操作。

- AddString

调用该函数以向组合框的列表框选项的末尾添加一个字符串(选项),如果组合框具有 CBS_SORT 风格,则该字符串将被添加到排序后的相应位置,该函数的原型为:

```
int AddString( LPCTSTR lpszString );
```

返回值:

如果函数调用成功,则返回所添加字符串在组合框中的索引,否则返回 CB_ERR。如果没有足够的空间存放新添加的字符串,则返回 CB_ERRSPACE。

参数:

lpszString —— 指定了将添加的字符串。

- DeleteString

调用该函数以从组合框中删除指定字符串(选项),其原型为:

```
int DeleteString( UINT nIndex );
```

返回值:

如果函数调用成功,则返回组合框中尚存的选项数目,否则返回 CB_ERR。

参数：

nIndex —— 指定了将被删除的字符串的索引。

- InsertString

调用该函数以向组合框的列表框插入一个选项,其原型为:

```
int InsertString( int nIndex, LPCTSTR lpszString );
```

返回值：

如果函数调用成功,则返回所插入字符串在组合框中的索引,否则返回 CB_ERR。如果没有足够的空间存放新插入的字符串,则返回 CB_ERRSPACE。

参数：

nIndex —— 指定了字符串将插入的位置(索引),如果该参数为 -1 则将字符串添加到列表框选项的末尾。

lpszString —— 指定了将插入的字符串。

- ResetContent

调用该函数以删除组合框中的所有选项(包括列表框和编辑控件),其原型为:

```
void ResetContent( );
```

- Dir

调用该函数以向组合框添加一个文件名列表,其原型为:

```
int Dir( UINT attr, LPCTSTR lpszWildCard );
```

返回值：

如果函数调用成功,则返回所添加的最后一个文件名在组合框中的索引,否则返回 CB_ERR。如果没有足够的空间存放新插入的文件名,则返回 CB_ERRSPACE。

参数：

attr —— 该参数可以为 CFile::GetStatus 函数得到的任何 enum 值的组合,或者为表 4-5 中所示值的组合。

表 4-5 attr 参数取值

attr 参数取值	含义
DDL_READWRITE	文件可以被读出或写入
DDL_READONLY	文件只读
DDL_HIDDEN	文件被隐藏,不显示在目录列表中
DDL_SYSTEM	文件为系统文件
DDL_DIRECTORY	lpszWildCard 参数指定的为目录名
DDL_ARCHIVE	文件可以被归档
DDL_DRIVES	包括所有符合 lpszWildCard 参数的驱动器
DDL_EXCLUSIVE	独占标志,如果设置了该标志,只有指定类型的文件被列出。否则,除正常文件外,还列出指定类型的文件

`lpszWildcard` —— 为将添加的文件名字符串。在字符串中可以包括通配符。

- `FindString`

调用该函数以检索列表框中第一个包含指定前缀的字符串,其原型为:

```
int FindString( int nStartAfter, LPCTSTR lpszString ) const;
```

返回值:

如果检索成功,则返回所找到的字符串在组合框中的索引,否则返回 `CB_ERR`。

参数:

`nStartAfter` —— 指定了检索的开始位置(索引)。当检索到列表框末尾时,将继续进行检索,直到到达由 `nStartAfter` 指定的选项。如果该参数为 `-1`,则将从头检索。

`lpszString` —— 指定了将检索的字符串前缀。检索操作与字符的大小写有关,因此该参数中可以包括任何大小写字母的组合。

- `FindStringExact`

调用该函数以在组合框中检索符合指定字符串的选项,其原型为:

```
int FindStringExact( int nIndexStart, LPCTSTR lpszFind ) const;
```

返回值:

如果检索成功,则返回符合条件的选项的索引,否则返回 `CB_ERR`。

参数:

`nIndexStart` —— 指定了检索的开始位置(索引)。当检索到列表框末尾时,将继续进行检索,直到到达由 `nStartAfter` 指定的选项。如果该参数为 `-1`,则将从头检索。

`lpszFind` —— 指定了将检索的字符串。检索操作与字符的大小写有关,因此该参数中可以包括任何大小写字母的组合。

如果组合框具有 `Owner-Draw` 风格,并且没有指定 `CBS_HASSTRINGS` 风格,则 `FindStringExact` 将视图使用双字值进行匹配。

- `SelectString`

调用该函数以选择组合框中的某选项,并将其显示在编辑控件中,其原型为:

```
int SelectString( int nStartAfter, LPCTSTR lpszString );
```

返回值:

如果检索成功,则返回符合条件的选项的索引,否则返回 `CB_ERR`,并且不改变当前选项。

参数:

`nStartAfter` —— 指定了检索的开始位置(索引)。当检索到列表框末尾时,将继续进行检索,直到到达由 `nStartAfter` 指定的选项。如果该参数为 `-1`,则将从头检索。

`lpszString` —— 指定了将检索的字符串前缀。检索操作与字符的大小写有关,因此该参数中可以包括任何大小写字母的组合。

`SelectString` 和 `FindString` 成员函数都进行字符串检索,但是 `SelectString` 成员函数将同时选择所找到的字符串。

4.1.5 重载函数

CComboBox 类的重载函数包括：DrawItem、MeasureItem、CompareItem 和 DeleteItem 函数，它们能够完成在组合框中绘制、比较和删除等操作。

• DrawItem

当组合框的任何可见部分发生改变时，框架调用该函数重绘组合框，其原型为：

```
virtual void DrawItem( LPDRAWITEMSTRUCT lpDrawItemStruct );
```

参数：

lpDrawItemStruct —— 为指向 DRAWITEMSTRUCT 结构的指针，其中包含了绘制所需的信息，具体参见 2.1 节。

DRAWITEMSTRUCT 结构中的 itemAction 成员定义了将进行的绘制操作。

默认情况下，该成员函数不作任何事。用户可以重载该函数以实现定制的绘制操作。在成员函数结束前，应用程序应该恢复先前的 GDI 对象。这一点非常重要，否则有可能出现异常情况。

• MeasureItem

在创建 Owner-Draw 型组合框时，框架调用该函数以确定组合框的尺寸，其原型为：

```
virtual void MeasureItem( LPMEASUREITEMSTRUCT lpMeasureItemStruct );
```

参数：

lpMeasureItemStruct —— 为指向 MEASUREITEMSTRUCT 结构的长型指针，该结构的定义如下：

```
typedef struct tagMEASUREITEMSTRUCT {
    UINT    CtlType;
    UINT    CtlID;
    UINT    itemID;
    UINT    itemWidth;
    UINT    itemHeight;
    DWORD   itemData
} MEASUREITEMSTRUCT;
```

结构成员：

CtlType —— 指定了控件类型，其取值如表 4-6 所示。

表 4-6 CtlType 成员取值

CtlType 成员取值	含义
ODT_COMBOBOX	自绘制组合框
ODT_LISTBOX	自绘制列表框
ODT_MENU	自绘制菜单项

CtrlID —— 指定了需要自绘制的控件 ID,而对于菜单项则无需使用该成员。

itemID —— 可以为菜单项 ID、列表框或组合框中某项的索引。对于空列表框或组合框,该成员为负值,这时应用程序只绘制焦点矩形(其坐标由 **rcItem** 成员给出)。虽然此时控件中没有需要显示的项,但绘制焦点矩形还是很有必要的,因为这能够提示用户该控件是否具有输入焦点。当然,也可以设置 **itemAction** 成员为合适的值,使得无需绘制输入焦点。

itemWidth —— 指定了菜单项的宽度,自绘制菜单项的父窗口必须在通告返回时,设置该成员。

itemHeight —— 指定了菜单项的高度,自绘制菜单项的父窗口必须在通告返回时,设置该成员。

itemData —— 对于列表框或组合框,该成员的取值可以为由 **CComboBox::AddString**、**CComboBox::InsertString**、**CListBox::AddString** 或 **CListBox::InsertString** 等函数传递给控件的值。

对于菜单项,该成员取值可以为由 **CMenu::AppendMenu**、**CMenu::InsertMenu** 或 **CMenu::ModifyMenu** 等函数传递给菜单的值。

默认情况下,该成员函数不作任何事。用户可以重载该函数,并填充 **MEASUREITEMSTRUCT** 结构,以将组合框的列表框尺寸通知 Windows。如果组合框具有 **CBS_OWNERDRAWVARIABLE** 风格,则框架将为列表框中的每个选项调用该函数。否则,该成员函数只需被调用一次。

- **CompareItem**

框架调用该函数以确定新选项在组合框中的相对位置,其原型为:

```
virtual int CompareItem( LPCOMPAREITEMSTRUCT lpCompareItemStruct );
```

返回值:

如果函数调用成功,则返回在 **COMPAREITEMSTRUCT** 结构中描述的两项的相对位置,可能的返回值如表 4-7 所示:

表 4-7 **CompareItem** 函数的返回值

返回值	含义
-1	第一项排在第二项之前
0	第一项与第二项的顺序相同
1	第一项排在第二项之后

参数:

lpCompareItemStruct —— 为指向 **COMPAREITEMSTRUCT** 结构的长型指针,该结构的定义如下:

```
typedef struct tagCOMPAREITEMSTRUCT {
    UINT    CtlType;
    UINT    CtlID;
```

```

    HWND    hwndItem;
    UINT    itemID1;
    DWORD    itemData1;
    UINT    itemID2;
    DWORD    itemData2;
} COMPAREITEMSTRUCT;

```

成员：

CtlType —— 指定了控件的类型,其取值可以为 **ODT_LISTBOX**(自绘制列表框)或 **ODT_COMBOBOX**(自绘制组合框)。

CtlID —— 指定了列表框或组合框的控件 ID。

HwndItem —— 指定了控件的父窗口句柄。

itemID1 —— 指定了在列表框或组合框中要比较的第一项。

itemData1 —— 应用程序为第一项提供的数据。在向组合框或列表框中时添加该项的调用中将使用该值。

itemID2 —— 指定了在列表框或组合框中要比较的第二项。

itemData2 —— 应用程序为第二项提供的数据。在向组合框或列表框中时添加该项的调用中将使用该值。

COMPAREITEMSTRUCT 结构为自绘制列表框或组合框中的排序项提供了标识符和数据。对于具有 **LBS_SORT** 或 **CBS_SORT** 风格的自绘制列表框或组合框,当应用程序向其中添加新项时,Windows 将向其父窗口发送 **WM_COMPAREITEM** 消息。该消息的 **lParam** 参数包含了指向 **COMPAREITEMSTRUCT** 结构的指针。当收到该消息后,父窗口将比较这两项,并返回排序顺序。

默认情况下, **CompareItem** 函数不作任何事。对于具有 **LBS_SORT** 风格的自绘制组合框,必须重载该成员函数以辅助框架对添加项的排序。

- **DeleteItem**

当从自绘制组合框中删除某项时,框架会调用该函数,其原型为:

```
virtual void DeleteItem( LPDELETEITEMSTRUCT lpDeleteItemStruct );
```

参数：

lpDeleteItemStruct —— 为指向 **DELETEITEMSTRUCT** 结构的长型指针,该结构包含了被删除项的信息,其定义如下:

```

typedef struct tagDELETEITEMSTRUCT {
    UINT CtlType;
    UINT CtlID;
    UINT itemID;
    HWND hwndItem;
    UINT itemData;
} DELETEITEMSTRUCT;

```

结构成员：

指定了控件的类型,其取值可以为 **ODT_LISTBOX**(自绘制列表框)或 **ODT_COM-**

BOBOX(自绘制组合框)。

CtlID —— 指定了列表框或组合框的控件 ID。

itemID —— 指定了列表框或组合框中将被删除的项的 ID。

hwndItem —— 指定了控件标识符。

itemData —— 应用程序定义的项数据。

默认情况下,该函数不作任何事。用户可以重载该函数以定制对列表框或组合框的重绘。

4.2 改变组合框控件的行为

改善用户界面的一个很重要的方面,就是使界面对用户更加友好。本节将向读者介绍,怎样改变组合框控件的默认行为,以提高控件的易用性。

4.2.1 自动完成组合框控件

所谓自动完成就是指只需输入部分选项,例如输入选项的起始字符,控件就会自动选择具有相同起始字符的选项。我们在 5.1 节中向读者介绍过,组合框有 3 种类型:下拉列表式、下拉式和简单型。MFC 为这三种组合框控件提供的支持并不完全相同。例如,下拉列表式组合框支持自动完成风格,而下拉式组合框则不支持这种风格。而本节的目的就是使下拉式组合框控件也具备这个功能。

当用户在组合框中输入一个键时,控件将选择第一个以该字符开头的选项。而当用户继续输入时,控件也同步检查并选择合适的选项。如果控件中并没有与用户最终输入的字符串符合的选项,则将自动根据用户输入创建新选项。

完成这一功能有好几种方式,这里我们将使用归类(subclassing)方法。使用归类能够修改 Windows 控件,并为其提供所需的功能,这实际与继承有些相似。例如,如果需要归类一个编辑控件,而此编辑控件只接受数字、'.'、Delete 和 Backspace 键的输入,则首先创建一个新的从 CEdit 派生的类,并使用该类在应用程序中创建控件。假设在对话框中,则首先在类的声明中添加下述代码:

```
CnumericEdit m_xAmount;
```

然后在 OnInitDialog 函数中添加如下代码:

```
m_xAmount.SubclassDlgItem(IDC_AMOUNT, this);
```

其中 IDC_AMOUNT 为编辑控件的 ID。

归类这样编辑控件之类的控件当然很简单,然而由于组合框是由编辑框和列表框组成的,处理可就没这么容易了。如需归类,则必须对编辑框部分和列表框部分分别归类。出于这个考虑,下面创建了两个类: CComboBox 的派生类 CTypeAheadCombo 和 CEdit 的派生类 CComboEdit。之所以要创建组合框控件的派生类,就是要将组合框归类以使其能够

使用归类后的编辑控件。

首先,我们看看 CTypeAheadCombo 类的定义,如清单 4-1 所示:

清单 4-1 CTypeAheadCombo 类的定义

```
class CTypeAheadCombo : public CComboBox
{
public:
    CTypeAheadCombo();

public:
    CComboEdit m_xEdit;

    //{AFX_VIRTUAL(CTypeAheadCombo)
    protected:
        virtual void PreSubclassWindow();
    //{AFX_VIRTUAL
public:
    virtual ~CTypeAheadCombo();
protected:
    //{AFX_MSG(CTypeAheadCombo)
    //{AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

通过该类的定义可以看出,CTypeAheadCombo 类中实际只定义了一个编辑控件成员 m_xEdit,和重载了 PreSubclassWindow 函数。其中前者是具有新功能的编辑控件类,而后者则负责使组合框的编辑控件部分归类于新的编辑控件类。清单 4-2 所示为 PreSubclassWindow 函数的源代码:

清单 4-2 PreSubclassWindow() 函数

```
void CTypeAheadCombo::PreSubclassWindow()
{
    m_xEdit.SubclassDlgItem(1001, this);
    CComboBox::PreSubclassWindow();
}
```

在 SubclassDlgItem 函数中,第一个参数为将被归类的控件 ID。这就出现了另一个问题:组合框控件无疑是有 ID 的,那么其中的组成部分有 ID 吗?如果有又如何得到呢?幸运的是,组合框中的编辑框部分的 ID 总是被分配为 1001。因此,只要将 SubclassDlgItem 函数的第一个参数设置为 1001 就一切 OK 了。

接下来就该介绍实现控件新功能的主要部分: CComboEdit 类。清单 4-3 所示为 CComboEdit 类的定义:

清单 4-3 CComboEdit 类的定义

```
class CComboEdit : public CEdit
{
// Construction
```



```

public:
    CComboEdit();

// Attributes
protected:
    int            m_nStartAfter;
    bool           m_bHighlighted;
    bool           m_bBackspace;
    bool           m_bDelete;
    CString        m_SearchString;
    WORD           m_nPos;

// Operations
protected:
    void DoFind(UINT nChar);

// Overrides
    // ClassWizard generated virtual function overrides
    //{AFX_VIRTUAL(CComboEdit)
    //{AFX_VIRTUAL

// Implementation
public:
    virtual ~CComboEdit();

    // Generated message map functions
protected:
    //{AFX_MSG(CComboEdit)
    afx_msg void OnChar(UINT nChar, UINT nRepCnt, UINT nFlags);
    afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
    //{AFX_MSG

    DECLARE_MESSAGE_MAP()
};

```

其中定义的成员变量主要是作为标志使用的。例如, `m_bBackspace` 和 `m_bDelete` 分别表示是否按下了 Backspace 和 Delete 键。为了能使组合框控件对用户的每一次输入都作出响应,那么显然应该在 `CComboEdit` 类中重载 `OnChar` 函数。通过该函数同步捕获用户输入的字符,就能创建“搜索字符串”(`m_SearchString`)用于定位组合框选项。清单 4-4 所示为 `OnChar` 函数的源代码:

清单 4-4 OnChar() 函数

```

void CComboEdit::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    // 当不是按下 DELETE 键时,调用基类的函数进行处理
    if (nChar != VK_DELETE)
        CEdit::OnChar(nChar, nRepCnt, nFlags);

    if (m_bHighlighted) {
        if ((m_bBackspace) || (m_bDelete)) {
            // 当选项被高亮显示时,按下 Backspace 或 Delete 都会将选项删除

```

```

        m_SearchString.Empty();
        m_bHighlighted = false;
        SetWindowText("");
        m_nStartAfter = -1;
    }
    else {
        // 如果输入的是字符,则进行寻找
        DoFind(nChar);
    }
}
else {
    // 如果没有高亮显示的选项,也进行寻找
    DoFind(nChar);
}
}

```

OnChar 函数对用户的输入作出判断: 如果在有选项被选中时按下 Backspace 或 Delete 键, 则清除该选项, 并同时清空“搜索字符串”; 否则, 调用 DoFind 函数寻找匹配选项。清单 4-5 所示为 DoFind 函数的源代码:

清单 4-5 DoFind() 函数

```

void CComboEdit::DoFind(UINT nChar)
{
    CString temp;
    int length = 0, index = 0;
    CTypeAheadCombo * pParent = (CTYPEAheadCombo *) GetParent();

    // 得到用户的输入
    GetWindowText(temp);
    // 将其添加到“搜索字符串”末尾
    m_SearchString += temp;
    // 得到新字符串的长度
    length = m_SearchString.GetLength();
    if (length) {
        // 搜索字符串中应该至少有一个字符(长度不为 0)
        index = pParent->FindString(m_nStartAfter, m_SearchString);
        if (index != -1) {
            // 如果找到匹配选项,则高亮显示
            pParent->SetCurSel(index);
            m_nStartAfter = index;
            m_bHighlighted = true;
        }
    }
    else {
        // 否则,重新设置搜索字符串,并设置合适的光标
        if (m_bBackspace) {
            // 如果按下了 Backspace 键,则光标位置要比原来位置退后一个字符
            pParent->SetEditSel(m_nPos - 1, m_nPos - 1);
        }
        else if (m_bDelete) {

```

```

        // 如果按下 DELETE 键,则没有变化
        pParent->SetEditSel(m_nPos, m_nPos);
    }
    else if (m_bHighlighted) {
        // 如果字符串已高亮显示,
        // 用新字符串替换它并把光标置于末尾
        SetWindowText(m_SearchString);
        pParent->SetEditSel(length, length);
    }
    else {
        // 如果没有高亮显示,则将光标移走
        pParent->SetEditSel(m_nPos + 1, m_nPos + 1);
    }
    // 清除选择字符串
    m_SearchString.Empty();
    m_bHighlighted = false;
}
}
}

```

上述代码的基本原理就是,取得用户的输入并将其添加到“搜索字符串”中,然后调用父窗口(组合框)的成员函数 FindString 寻找匹配选项。

这时读者可能已经感到,寻找匹配字符串并不难,真正的难点在于如何处理其他键盘输入。例如,如果用户使用箭头键改变光标位置后再进行输入。而 OnChar 函数只能跟踪输入的新字符,而无法取得光标所在位置。另外,当用户按下 Backspace 键时,会删除输入编辑框中的部分文本。此时问题在于,如何能判断是否应该删除文本的一部分,还是全部删除文本(选项出于选中状态)并清空编辑框。此时,就需要响应键盘 WM_KEYDOWN 消息,以跟踪光标、Delete 等键的输入。清单 4-6 所示为 OnKeyDown 函数的源代码:

清单 4-6 OnKeyDown()函数

```

void CComboEdit::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    CEdit::OnKeyDown(nChar, nRepCnt, nFlags);
    CTypeAheadCombo * pParent = (CTypeAheadCombo *) GetParent();

    // 如果按下 Delete 或 Backspace 键,则设置标志
    m_bDelete = false;
    m_bBackspace = false;
    switch (nChar) {
    case VK_DELETE:
        m_bDelete = true;
        break;
    case VK_BACK:
        m_bBackspace = true;
        break;
    }
}

```

```

// 得到光标位置
m_nPos = LOWORD(pParent->GetEditSel());
if ((nChar == VK_LEFT) || (nChar == VK_RIGHT)) {
    // 如果左右箭头键被按下,则取消高亮显示
    m_bHighlighted = false;
    m_SearchString.Empty();
}

// 如果按下 Delete 键,则不会产生 WM_CHAR 消息,应该手动调用 OnChar
if (m_bDelete)
    OnChar(nChar, nRepCnt, nFlags);
}

```

上面的实现方法非常直观,因此很好理解,但是代码效率却不高。而且整个程序使用了很多标志,很容易导致混乱。清单 4-7 和 4-8 是改进后的 `OnKeyDown` 和 `OnChar` 函数。其中不需使用任何标记变量,而且无需调用 `DoFind` 函数,读者应该好好比较一下。

清单 4-7 OnKeyDown() 函数

```

void CComboEdit::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    CEdit::OnKeyDown(nChar, nRepCnt, nFlags);
    CSearchBox* pParent = (CSearchBox*) GetParent();

    if(nChar == VK_RETURN)
    {
        CString str;
        GetWindowText(str);
        pParent->AddString(str);
    }
}

```

清单 4-8 OnChar() 函数

```

void CComboEdit::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    CEdit::OnChar(nChar, nRepCnt, nFlags);
    int index, len;
    CString str;
    CSearchBox* pParent = (CSearchBox*) GetParent();
    if(nChar != VK_BACK)
    {
        GetWindowText(str);
        len = str.GetLength();
        index = pParent->FindString(-1, str);
        if(index != -1) //found something
        {
            pParent->SetCurSel(index);
            GetWindowText(str);
            pParent->SetEditSel(len, str.GetLength());
        }
    }
}

```


4.2.2 使用工具窗口替代列表框

对于下拉列表式或下拉式组合框控件,当用户单击其右边的按钮时,就会出现包含选项列表的下拉列表框。经过本书学习的读者此时可能就会有这样的想法:为什么不将包含选项列表的窗口换一种形式呢?本节就将向读者介绍如何使用下拉工具窗口代替原来的下拉列表窗口。

当然,设计的第一步就是由 CComboBox 派生一个新类: CCoolComboDlg。那么此类中应该作些什么工作呢?首先,当用户单击组合框的箭头按钮时,就会产生 ON_CBN_DROPDOWN 消息。而我们的目的就是要修改按下箭头按钮时出现的窗口,因此显然应该定制该消息的处理函数。清单 4-9 所示为 OnDropDownCombo 函数的源代码:

清单 4-9 OnDropDownCombo() 函数

```
void CCoolComboDlg::OnDropDownCombo()
{
    // 禁止显示下拉列表框
    m_CoolCombo.ShowDropDown(FALSE);
    // 显示我们自己的下拉窗口
    DisplayPopdownWindow();
}
```

在 OnDropDownCombo 消息处理函数中,首先以 FALSE 参数调用 ShowDropDown 函数以禁止下拉列表框的显示;然后调用 DisplayPopdownWindow 函数显示定制的窗口。DisplayPopdownWindow 函数的源代码如清单 4-10 所示:

清单 4-10 DisplayPopdownWindow() 函数

```
void CCoolComboDlg::DisplayPopdownWindow()
{
    // 得到控件尺寸
    CRect rectCombo;
    m_CoolCombo.GetWindowRect(&rectCombo);

    // 如果窗口已经存在,则删除之
    if(m_pWndPopDown)
        delete m_pWndPopDown;

    // 注册并创建新窗口
    LPCTSTR lpszClass = AfxRegisterWndClass(CS_HREDRAW|CS_VREDRAW);
    m_pWndPopDown = new CPopDownWnd();
    m_pWndPopDown->CreateEx(WS_EX_TOPMOST |
        WS_EX_TOOLWINDOW|WS_EX_PALETTEWINDOW
        lpszClass, _T(""), WS_POPUP | WS_VISIBLE,
```

```

rectCombo.left,rectCombo.bottom,
rectCombo.Width(),200,
m_CoolCombo.GetSafeHwnd(),NULL, NULL);
}

```

在创建窗口时,通过指定 `WS_EX_TOOL_WINDOW` 风格创建工具窗口。

现在似乎已经完成了所有的工作,然而问题依然存在:创建的工具窗口并不会与其父窗口(例如对话框)共同移动。一个最简单的方法就是禁止用户移动控件的父窗口。有好几种方法能够达到这个目的,此处将使用的是处理非客户区消息。这是因为在移动窗口时,需要使用鼠标拖动窗口的标题栏(非客户区)。清单 4-11 所示为 `OnNcHitTest` 函数的源代码:

清单 4-11 `OnNcHitTest()` 函数

```

UINT CCoolComboDlg::OnNcHitTest(CPoint point)
{
    UINT nHitTest = CDialog::OnNcHitTest(point);
    return (nHitTest == HTCAPTION) ? HTNOWHERE : nHitTest;
}

```

4.2.3 鼠标敏感组合框控件

第四章曾经向读者介绍过鼠标敏感编辑控件的创建方法,出于同样的理由,本节将介绍如何创建外观会随鼠标移动而改变的组合框控件。

实际上无论是编辑控件还是组合框控件的外观都是由其边框的绘制方法决定的。平坦外观的组合框控件实际没有边框,而 3D 外观的组合框控件则具有 3D 边框。下面将创建一个 `CHotComboBox` 类(`CComboBox` 的派生类),它将负责管理组合框控件边框的绘制。清单 4-12 所示为 `CHotComboBox` 类的定义:

清单 4-12 `CHotComboBox` 类的定义

```

class CHotComboBox : public CComboBox
{
// Construction
public:
    CHotComboBox();

// Attributes
public:

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{AFX_VIRTUAL(CHotComboBox)
    //{AFX_VIRTUAL

// Implementation

```

```

public:
    virtual ~CHotComboBox();
    // Generated message map functions
protected:
    virtual void DrawBorder(bool fHot = true);
    COLORREF m_clr3DHilight;
    COLORREF m_clr3DLight;
    COLORREF m_clr3DDkShadow;
    COLORREF m_clr3DShadow;
    COLORREF m_clr3DFace;
    bool m_fGotFocus;
    bool m_fTimerSet;
    //||AFX_MSG(CHotComboBox)
    afx_msg void OnPaint();
    afx_msg void OnSetFocus(CWnd* pOldWnd);
    afx_msg void OnKillFocus(CWnd* pNewWnd);
    afx_msg void OnMouseMove(UINT nFlags, CPoint point);
    afx_msg void OnTimer(UINT nIDEvent);
    afx_msg void OnNcMouseMove(UINT nHitTest, CPoint point);
    afx_msg void OnSysColorChange();
    afx_msg void OnSetfocus();
    afx_msg void OnKillfocus();
    //||AFX_MSG

    DECLARE_MESSAGE_MAP()
};

```

其中 COLORREF 类型的变量用于设置不同的边框颜色;m_fTimerSet 用于标识是否设置了定时器;m_fGotFocus 用于标识控件是否具有焦点。

由于控件边框的绘制是由鼠标动作引起的,因此要在类中处理鼠标移动消息,以设置控件边框显示与否。清单 4-13 所示为 OnMouseMove 函数的源代码:

清单 4-13 OnMouseMove()函数

```

void CHotComboBox::OnMouseMove(UINT nFlags, CPoint point)
{
    if (! m_fGotFocus) {
        if (! m_fTimerSet) {
            DrawBorder();
            SetTimer(1, 10, NULL);
            m_fTimerSet = true;
        }
    }

    CComboBox::OnMouseMove(nFlags, point);
}

```

如果控件具有输入焦点,当然控件的外观不需改变。因此在上述代码中,首先检查控

件是否具有输入焦点。如果控件没有输入焦点,则根据鼠标位置绘制按钮边框。这是通过定时器响应函数完成的。清单 4-14 所示为 OnTimer 函数:

清单 4-14 OnTimer()函数

```
void CHotComboBox::OnTimer(UINT nIDEvent)
{
    POINT pt;
    GetCursorPos(&pt);
    CRect rcItem;
    GetWindowRect(&rcItem);

    if(! rcItem.PtInRect(pt)) {
        KillTimer(1);

        m_fTimerSet = false;

        if (! m_fGotFocus) {
            DrawBorder(false);
        }
        return;
    }

    CComboBox::OnTimer(nIDEvent);
}
```

除了鼠标位置会影响控件外观外,控件是否具有焦点也会影响其外观。因此还应该处理 WM_SETFOCUS 和 WM_KILLFOCUS 消息。清单 4-15 和 4-16 所示为 OnSetFocus 和 OnKillFocus 函数的源代码:

清单 4-15 OnSetFocus()函数

```
void CHotComboBox::OnSetFocus(CWnd * pOldWnd)
{
    CComboBox::OnSetFocus(pOldWnd);
    m_fGotFocus = true;
    DrawBorder();
}
```

清单 4-16 OnKillFocus()函数

```
void CHotComboBox::OnKillFocus(CWnd * pNewWnd)
{
    CComboBox::OnKillFocus(pNewWnd);
    m_fGotFocus = false;
    DrawBorder(false);
}
```

另外,边框绘制操作被封装在 DrawBorder 函数中,其源代码如清单 4-17 所示:

清单 4-17 DrawBorder()函数

```
void CHotComboBox::DrawBorder(bool fHot)
{
}
```



```
CDC * pDC = GetDC();
CRect rcItem;
DWORD dwExStyle = GetExStyle();

GetWindowRect(&rcItem);
ScreenToClient(&rcItem);

if (! IsWindowEnabled()) {
    fHot = true;
}

if (dwExStyle & (WS_EX_STATICEDGE | WS_EX_DLGMODALFRAME)) {
    if (dwExStyle & WS_EX_STATICEDGE) {
        rcItem.DeflateRect(1, 1, 0, 0);
    }
    if (dwExStyle & WS_EX_DLGMODALFRAME) {
        rcItem.DeflateRect(1, 1, 0, 0);
    }
    rcItem.DeflateRect(1, 1, 0, 0);
} else {
    rcItem.DeflateRect(1, 1);
}

if (fHot) {
    pDC->Draw3dRect(rcItem, m_clr3DDkShadow, m_clr3DLight);
    rcItem.InflateRect(1, 1);
    pDC->Draw3dRect(rcItem, m_clr3DShadow, m_clr3DHilight);

    if (dwExStyle & WS_EX_DLGMODALFRAME) {
        pDC->Draw3dRect(rcItem, m_clr3DShadow, m_clr3DFace);
        rcItem.InflateRect(1, 1, 0, 0);
        if (dwExStyle & WS_EX_STATICEDGE) {
            rcItem.DeflateRect(0, 0, 2, 2);
        } else {
            rcItem.DeflateRect(0, 0, 1, 1);
        }
        pDC->Draw3dRect(rcItem, m_clr3DDkShadow, m_clr3DDkShadow);
    }

    if (dwExStyle & WS_EX_STATICEDGE) {
        if (dwExStyle & WS_EX_DLGMODALFRAME) {
            rcItem.InflateRect(1, 1);
        } else {
            rcItem.InflateRect(1, 1, 0, 0);
        }
        pDC->Draw3dRect(rcItem, m_clr3DShadow, m_clr3DHilight);
    }
} else {
    pDC->Draw3dRect(rcItem, m_clr3DFace, m_clr3DFace);
    rcItem.InflateRect(1, 1);
    pDC->Draw3dRect(rcItem, m_clr3DFace, m_clr3DFace);
}
```

```

        if (dwExStyle & WS_EX_DLGMODALFRAME) {
            rcItem.InflateRect(1, 1, 0, 0);
            if (dwExStyle & WS_EX_STATICEDGE) {
                rcItem.InflateRect(0, 0, 1, 1);
                pDC->Draw3dRect(rcItem, m_clr3DFace, m_clr3DFace);
                rcItem.DeflateRect(0, 0, 3, 3);
            } else {
                rcItem.DeflateRect(0, 0, 1, 1);
            }
            pDC->Draw3dRect(rcItem, m_clr3DFace, m_clr3DFace);
        }

        if (dwExStyle & WS_EX_STATICEDGE) {
            if (dwExStyle & WS_EX_DLGMODALFRAME) {
                rcItem.InflateRect(1, 1);
            } else {
                rcItem.InflateRect(1, 1, 0, 0);
            }
            pDC->Draw3dRect(rcItem, m_clr3DFace, m_clr3DFace);
        }
    }

    ReleaseDC(pDC);
}

```

在使用时,只要使编辑控件由 CHotEdit 类管理即可(例如,通过 ClassWizard 为编辑控件添加 CHotEdit 类型的变量)。图 4-2 所示即为鼠标敏感组合框控件。

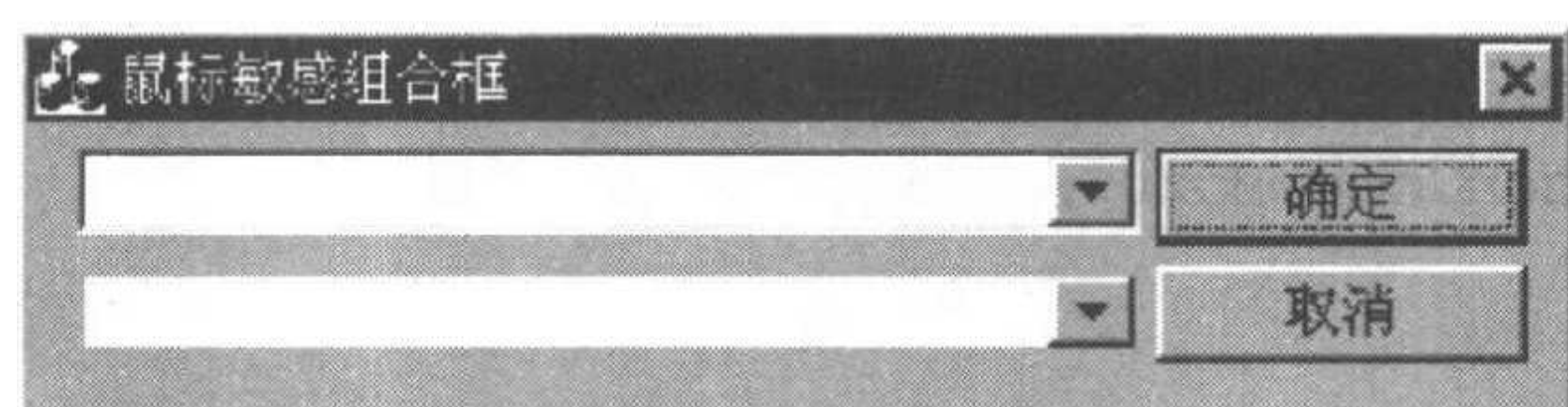


图 4-2 鼠标敏感组合框控件

4.3 改变组合框控件选项形式

一般情况下,组合框中的选项为字符串。这由 CComboBox 类中负责添加选项的函数 AddString 也可以看出。而对于希望使界面看起来与众不同的程序员来说,修改控件选项形式就是一个方法。

4.3.1 图标选择组合框控件

使用图标作为选项的例子有很多,例如 Windows 中更改图标的界面,如图 4-3 所示:本节将实现的效果如图 4-4 所示:

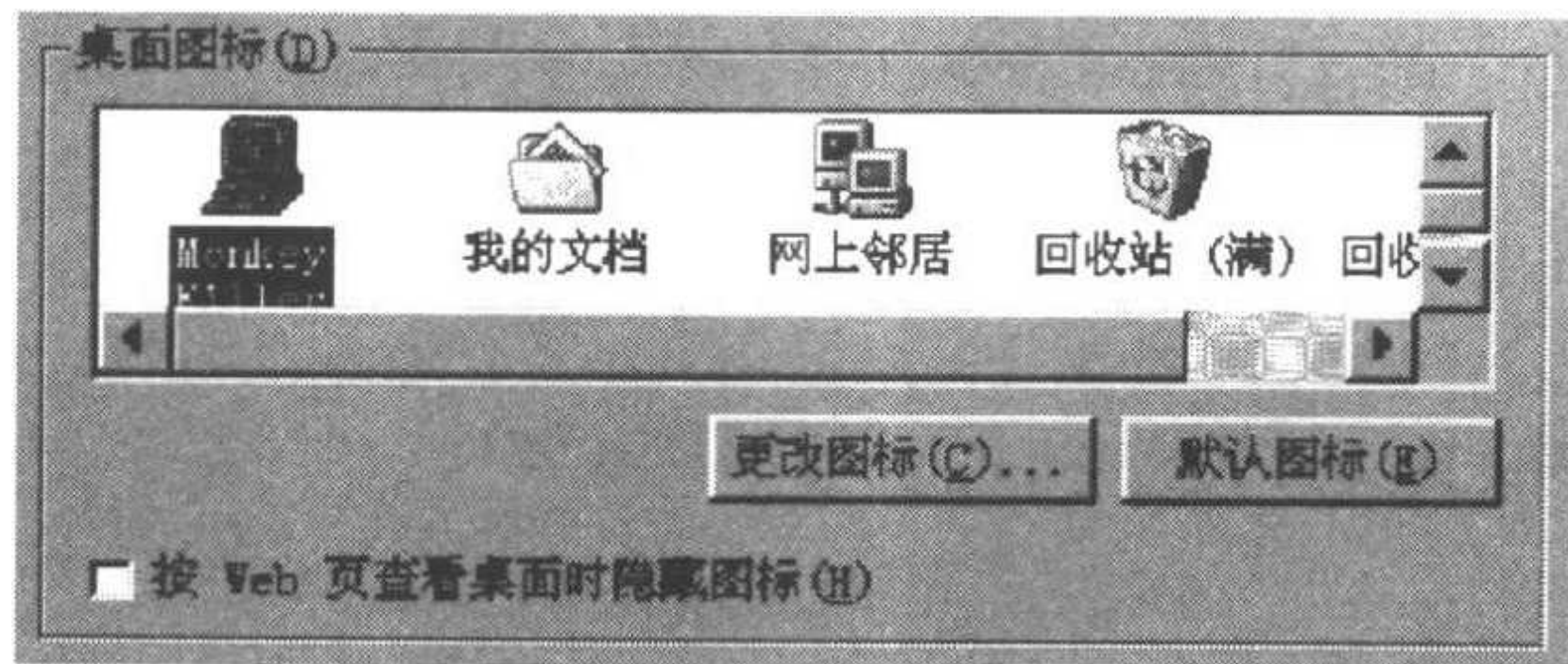


图 4-3 更改图标

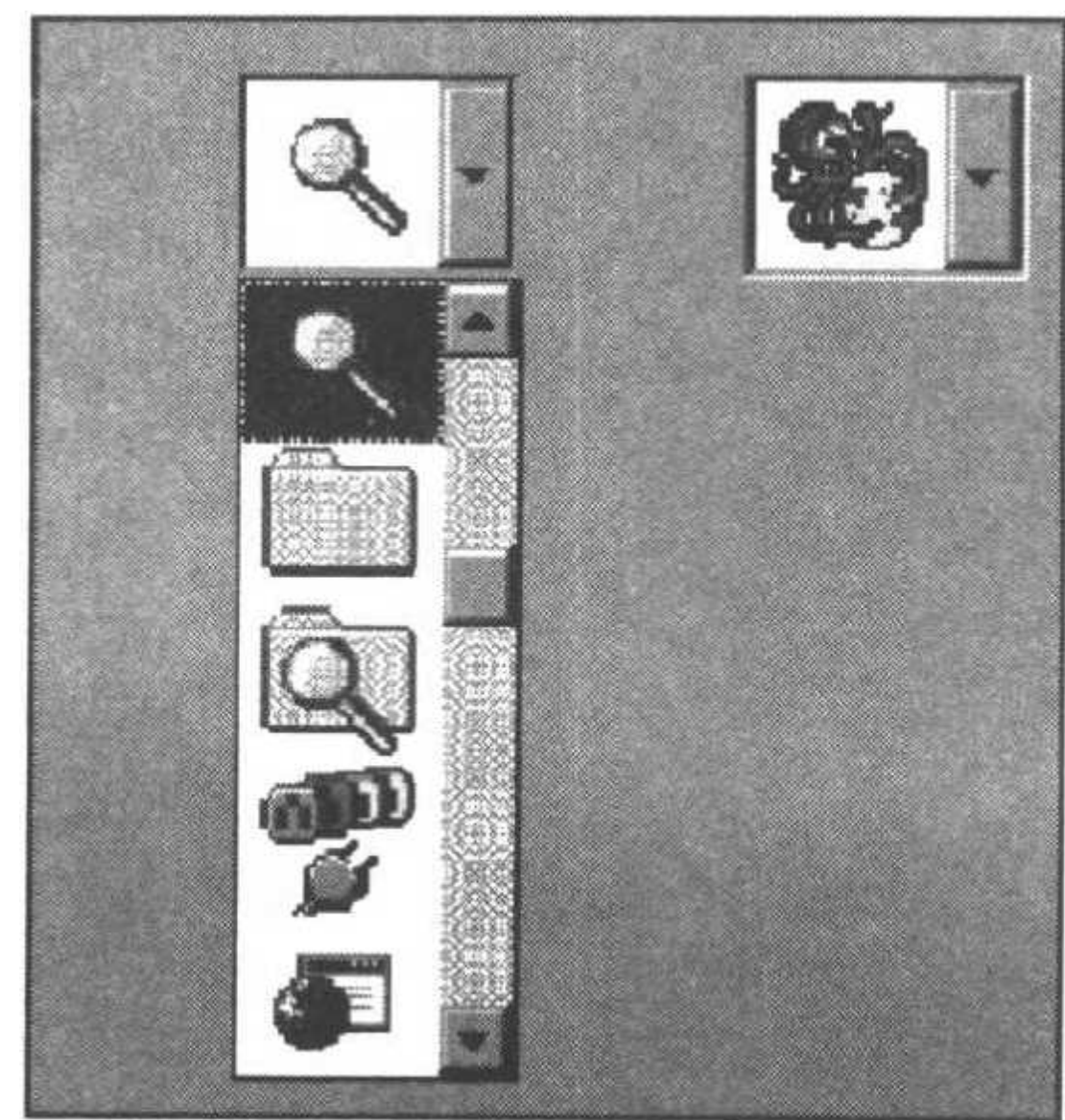


图 4-4 图标选择组合框控件

首先创建 CComboBox 的派生类 CIconComboBox, 用于管理图标选择组合框控件。接下来就需要设计其功能。由于将文本选项用图标进行了替换, 因此 CIconComboBox 类中首先应该重载 DrawItem 函数。要绘制图标, 就必须以合适的尺寸进行绘制, 因此需要重载 MeasureItem 函数。此外, CIconComboBox 类还必须完成图标选项的添加、删除、插入以及选择等功能。根据以上思路, CIconComboBox 类的定义如清单 4-18 所示:

清单 4-18 CIconComboBox 类的定义

```
class CIconComboBox : public CComboBox
{
// Construction/Destruction
public:
    CIconComboBox();
    virtual ~CIconComboBox();

// Attributes
public:
    CSize m_sizeIcon;

// Operations
public:
    virtual int AddIcon(LPCTSTR lpszIconFile);
    virtual int InsertIcon(int nIndex, LPCTSTR lpszIconFile);
    virtual int SelectIcon(LPCTSTR lpszIconFile);
    virtual int SelectIcon(int nIndex);
    virtual int DeleteIcon(LPCTSTR lpszIconFile);
    virtual int DeleteIcon(int nIndex);

// Implementation
protected:
    virtual void OnOutputIcon(LPDRAWITEMSTRUCT lpDIS, BOOL bSelected);

// Overrides
    virtual int AddString(LPCTSTR lpszString);
    virtual int InsertString(int nIndex, LPCTSTR lpszString);
    virtual int DeleteString(int nIndex);
};
```

```

virtual void MeasureItem(LPMEASUREITEMSTRUCT lpMIS);
virtual void DrawItem(LPDRAWITEMSTRUCT lpDIS);
}

```

下面就向读者介绍类的功能实现。首先从 DrawItem 函数开始,它负责完成图标选项的绘制,其源代码如清单 4-19 所示:

清单 4-19 DrawItem() 函数

```

void CIconComboBox::DrawItem(LPDRAWITEMSTRUCT lpDIS)
{
    CDC * pDC = CDC::FromHandle(lpDIS->hDC);

    if (! IsWindowEnabled())
    {
        CBrush brDisabled(RGB(192,192,192)); // 灰色
        CBrush * pOldBrush = pDC->SelectObject(&brDisabled);
        CPen penDisabled(PS_SOLID, 1, RGB(192,192,192));
        CPen * pOldPen = pDC->SelectObject(&penDisabled);
        OnOutputIcon(lpDIS, FALSE);
        pDC->SelectObject(pOldBrush);
        pDC->SelectObject(pOldPen);
        return;
    }

    // 被选中
    if ((lpDIS->itemState & ODS_SELECTED)
        && (lpDIS->itemAction & (ODA_SELECT | ODA_DRAWENTIRE)))
    {
        CBrush brHighlight(::GetSysColor(COLOR_HIGHLIGHT));
        CBrush * pOldBrush = pDC->SelectObject(&brHighlight);
        CPen penHighlight(PS_SOLID, 1, ::GetSysColor(COLOR_HIGHLIGHT));
        CPen * pOldPen = pDC->SelectObject(&penHighlight);
        pDC->Rectangle(&lpDIS->rcItem);
        pDC->SetBkColor(::GetSysColor(COLOR_HIGHLIGHT));
        pDC->SetTextColor(::GetSysColor(COLOR_HIGHLIGHTTEXT));
        OnOutputIcon(lpDIS, TRUE);
        pDC->SelectObject(pOldBrush);
        pDC->SelectObject(pOldPen);
    }

    // 取消选择
    if (! (lpDIS->itemState & ODS_SELECTED)
        && (lpDIS->itemAction & (ODA_SELECT | ODA_DRAWENTIRE)))
    {
        CBrush brWindow(::GetSysColor(COLOR_WINDOW));
        CBrush * pOldBrush = pDC->SelectObject(&brWindow);
        CPen penHighlight(PS_SOLID, 1, ::GetSysColor(COLOR_WINDOW));
        CPen * pOldPen = pDC->SelectObject(&penHighlight);
        pDC->Rectangle(&lpDIS->rcItem);
        pDC->SetBkColor(::GetSysColor(COLOR_WINDOW));
    }
}

```



```

        pDC->SetTextColor(::GetSysColor(COLOR_WINDOWTEXT));
        OnOutputIcon(lpDIS, FALSE);
        pDC->SelectObject(pOldBrush);
        pDC->SelectObject(pOldPen);
    }

    // 焦点态
    if (lpDIS->itemAction & ODA_FOCUS)
    {
        pDC->DrawFocusRect(&lpDIS->rcItem);
    }
}

```

与前几章中的设计思想一样, DrawItem 函数首先需要取得控件选项的当前状态, 然后根据选项状态决定其绘制方式。DrawItem 函数在绘制时, 调用函数 OnOutputIcon, 该函数的源代码如清单 4-20 所示:

清单 4-20 OnOutputIcon() 函数

```

void CIconComboBox::OnOutputIcon(LPDRAWITEMSTRUCT lpDIS, BOOL bSelected)
{
    if (GetCurSel() == CB_ERR || GetCount() == 0)
        return; // 如果没有选项

    CDC * pDC = CDC::FromHandle(lpDIS->hDC);

    HICON hIcon = (HICON)lpDIS->itemData;
    ASSERT(hIcon != NULL);

    int x = lpDIS->rcItem.left + ((lpDIS->rcItem.right - lpDIS->rcItem.left)
                                   / 2) - (m_sizeIcon.cx / 2);
    int y = lpDIS->rcItem.top + ((lpDIS->rcItem.bottom - lpDIS->rcItem.top) /
                                   2) - (m_sizeIcon.cy / 2);

    pDC->DrawIcon(x, y, hIcon);
}

```

当 DrawItem 调用 OnOutputIcon 时, 将 LPDRAWITEMSTRUCT 结构变量 lpDIS 传递给它。该结构中包含了组合框控件中选项(此时为图标)的信息。OnOutputIcon 函数中首先由 lpDIS 结构得到控件的设备环境和尺寸信息, 然后据此调用设备环境类的成员函数 DrawIcon 绘制图标。

由于组合框中的选项为图标, 因此需要设置其合适的尺寸, 尤其是图标的高度。这是通过重载 MeasureItem 函数完成的, 源代码如清单 4-21 所示:

清单 4-21 MeasureItem() 函数

```

void CIconComboBox::MeasureItem(LPMEASUREITEMSTRUCT lpMIS)
{
    lpMIS->itemHeight = (m_sizeIcon.cy + 2);
}

```

类中其他函数主要用于操作组合框控件选项,这里以 AddIcon 为例,其源代码如清单 4-22 所示:

清单 4-22 AddIcon() 函数

```
int CIconComboBox::AddIcon(LPCTSTR lpszIconFile)
{
    HICON hIcon = ::ExtractIcon(AfxGetInstanceHandle(), lpszIconFile, 0);

    if (hIcon == (HICON)1 || hIcon == NULL)
        return CB_ERR;

    int nIndex = CComboBox::AddString(lpszIconFile);

    if (nIndex != CB_ERR && nIndex != CB_ERRSPACE)
        SetItemData(nIndex, (DWORD)hIcon);

    return nIndex;
}
```

在函数中调用组合框控件的 SetItemData 成员函数,将对应选项的附加数据设置为图标数据。这样在调用 DrawItem 时就会根据图标数据进行绘制,从而完成向组合框中添加和绘制图标选项的工作。其他 DeleteIcon、InsertIcon 等函数的功能与实现与 AddIcon 函数类似,这里就不再赘述了。

用户在使用 CIconComboBox 类时,只要将 IconComboBox.cpp 和 IconComboBox.h 包含在工程中。然后在资源编辑器中创建具有自绘制风格的下拉式组合框,并为该控件指定 CIconComboBox 类型的成员变量即可。此后在代码中就可以通过 AddIcon、DeleteIcon 等函数对控件进行操作了。

需要注意的一点是,要为组合框控件指定 CBS_HASSTRINGS 风格。这是因为组合框中实际放的是图标名称,以后对图标的选择或删除操作都是以图标名称为基础的。组合框控件的宽度是在创建时由代码决定的;而其高度可以由下面的语句进行手动设置:

```
m_cbMyIconComboBox.SetItemHeight(-1, m_cbMyIconComboBox.m_sizeIcon.cy + 6);
```

其中 m_sizeIcon.cy 为标准图标高度(SM_CYICON)。

4.3.2 字体选择组合框控件

字体选择组合框在编辑软件中是十分常见的,例如 Word。它一般存在于软件的工具栏中。本节向读者介绍如何创建字体选择组合框控件,如图 4-5 所示。而如何将其组合到工具栏中,会在第 7 章中进行介绍。

为了方便处理字体选项,首先设计一个字体管理类 CFontObj,其定义如清单 4-23 所示:

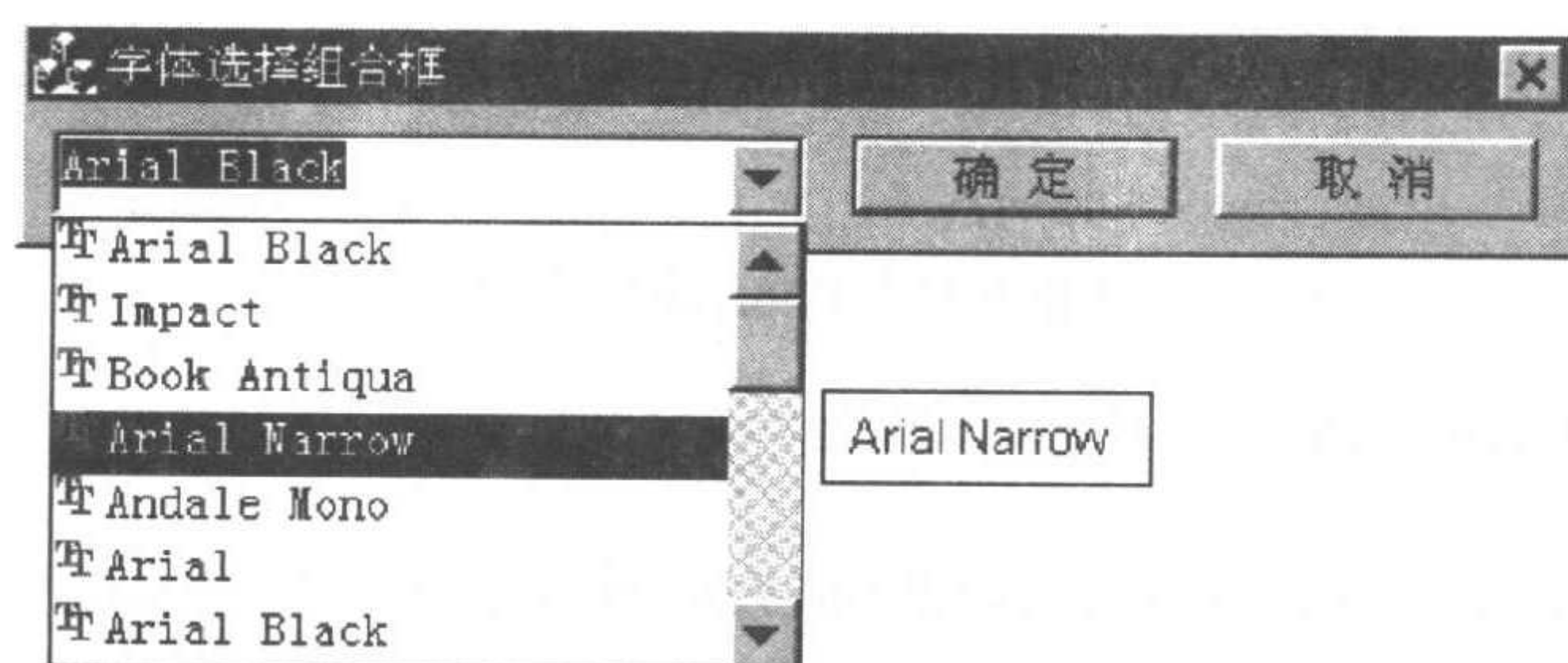


图 4-5 字体选择组合框

清单 4-23 CFontObj 类的定义

```

class CFontObj : public CObject
{
protected:
    BOOL m_bInUse; // 使用中的字体
    DWORD m_nFlags; // 字体标志
public:
    CFontObj(DWORD nFlags)
    {
        m_nFlags = nFlags;
        m_bInUse = FALSE;
    }

    CFontObj(CFontObj * pFontObj)
    {
        m_nFlags = pFontObj->GetFlags();
        m_bInUse = pFontObj->GetFontInUse();
    }

    int GetFlags() const { return m_nFlags; }
    BOOL GetFontInUse() { return m_bInUse; }
    void SetFontInUse(BOOL bInUse) { m_bInUse = bInUse; }
};

```

这个类非常简单,其成员函数的功能也直接在类定义中实现。其作用主要体现在两个成员变量 `m_bInUse` 和 `m_nFlags`,它们分别用于标识字体是否正在使用 and 字体标志。而类成员函数只是提供了外部获取成员变量值的接口。

接下来再设计字体选择组合框的管理类 `CFontCombo`,其定义如清单 4-24 所示:

清单 4-24 CFontCombo 类定义

```

class CFontCombo : public CComboBox
{
// Construction
public:
    CFontCombo();
    void Initialize();

protected:

```

```

    BOOL EnumerateFonts();
    void AddFont(CString strName, DWORD dwFlags);
    void SetCurrentFont();
    void SetFontInUse(const CString& strFont);
public:
    static BOOL CALLBACK AFX_EXPORT EnumFamScreenCallBackEx(
        ENUMLOGFONTEX * pelf, NEWTEXTMETRICEX * /* lpntm */, int FontType,
        LPVOID pThis);

    static BOOL CALLBACK AFX_EXPORT EnumFamPrinterCallBackEx(
        ENUMLOGFONTEX * pelf, NEWTEXTMETRICEX * /* lpntm */, int FontType,
        LPVOID pThis);

protected:
    CTipWnd          m_wndTip;
    CString          m_strFontSave;
    CString          m_strDefault;
    CImageList       m_img;

    CTypedPtrMap< CMapStringToPtr, CString, CFontObj * > m_mapFonts;
    CTypedPtrMap< CMapStringToPtr, CString, CFontObj * > m_mruFonts;

// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CFontCombo)
public:
    virtual void DeleteItem(LPDELETEITEMSTRUCT lpDeleteItemStruct);
    virtual void DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct);
    virtual void MeasureItem(LPMEASUREITEMSTRUCT lpMeasureItemStruct);
//}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CFontCombo();

    // Generated message map functions
protected:
    {{{AFX_MSG(CFontCombo)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnDestroy();
    afx_msg void OnKillfocus();
    afx_msg void OnSetfocus();
    afx_msg void OnCloseUp();
    afx_msg void OnTimer(UINT nIDEvent);
    }}}AFX_MSG

    DECLARE_MESSAGE_MAP()
};

```

由于组合框中的选项为字体,需要在创建组合框时从系统中读入字体信息,因此响应 WM_CREATE 函数对组合框内容进行初始化。清单 4-25 所示为 OnCreate 函数的源代码:

清单 4-25 OnCreate()函数

```
int CFontCombo::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CComboBox::OnCreate(lpCreateStruct) == -1)
        return -1;
    Initialize();
    return 0;
}
```

函数首先调用基类的 OnCreate 函数完成默认初始化,然后调用 Initialize 函数完成字体选项的初始化。清单 4-26 所示为 Initialize 函数的源代码:

清单 4-26 Initialize()函数

```
void CFontCombo::Initialize()
{
    // 创建提示窗口
    m_wndTip.Create(this);

    // 设置默认字体名称
    CString strDefault = "";

    CFontObj* pFontObj;
    CString strKey, strComp;
    EnumerateFonts();

    POSITION pos = m_mapFonts.GetStartPosition();
    while (pos)
    {
        m_mapFonts.GetNextAssoc(pos, strKey, pFontObj);
        int nMax = GetCount();
        BOOL bInsert = FALSE;
        for (int nIdx = 0; nIdx < nMax; nIdx++)
        {
            GetLBText(nIdx, strComp);
            if (strComp.Collate(strKey) == 1)
            {
                bInsert = TRUE;
                InsertString(nIdx, strKey);
                break;
            }
        }
        if (! bInsert)
            AddString(strKey);
    }

    SetTimer(1, 500, NULL);
}
```

在上述代码中,首先创建字体提示窗口。然后,调用 EnumerateFonts 函数枚举系统使用的字体。在得到了字体信息后,通过 m_mapFonts(字体映射对象,在后面还将进行介

绍)进行循环,将所有字体信息都输入组合框中。最后定时器监控选项的改变,用于显示提示窗口。清单 4-27 所示为 EnumateFonts 函数的源代码:

清单 4-27 EnumateFonts()函数

```

BOOL CFontCombo::EnumerateFonts()
{
    HDC hDC;
    // 得到屏幕字体
    hDC = ::GetWindowDC(NULL);
    LOGFONT lf;
    ZeroMemory(&lf, sizeof(lf));
    lf.lfCharSet = ANSI_CHARSET;

    if (! EnumFontFamiliesEx(hDC, &lf, (FONTENUMPROC) EnumFamScreenCallBackEx,
        (LPARAM) this, (DWORD) 0))
        return FALSE;
    ::ReleaseDC(NULL, hDC);
    // 得到打印机字体
    CPrintDialog dlg(FALSE);
    if (AfxGetApp() -> GetPrinterDeviceDefaults(&dlg.m_pd))
    {
        hDC = dlg.CreatePrinterDC();
        ASSERT(hDC != NULL);
        ZeroMemory(&lf, sizeof(lf));
        lf.lfCharSet = ANSI_CHARSET;
        if (! EnumFontFamiliesEx(hDC, &lf, (FONTENUMPROC) EnumFamPrinter-
            CallBackEx, (LPARAM) this, (DWORD) 0))
            return FALSE;
    }
    return TRUE;
}

```

需要注意的是,在获得屏幕字体和打印机字体时,EnumFontFamiliesEx 函数中使用了不同的回调函数。EnumFontFamiliesEx 能够枚举系统中所有与 LOGFONT 结构匹配的字体,其原型如下:

```

int EnumFontFamiliesEx( HDC hdc, LPLOGFONT lpLogfont,
    FONTENUMPROC lpEnumFontFamExProc, LPARAM lParam, DWORD dwFlags );

```

返回值:

如果函数调用成功,则返回回调函数的最后一个返回值。该值取决于指定设备可用的字体家族。

参数:

hdc —— 指定了设备环境。

lpLogfont —— 指定了包含字体信息的 LOGFONT 结构,该结构的定义如下:

```

typedef struct tagLOGFONT {
    LONG lfHeight;

```

```

    LONG lfWidth;
    LONG lfEscapement;
    LONG lfOrientation;
    LONG lfWeight;
    BYTE lfItalic;
    BYTE lfUnderline;
    BYTE lfStrikeOut;
    BYTE lfCharSet;
    BYTE lfOutPrecision;
    BYTE lfClipPrecision;
    BYTE lfQuality;
    BYTE lfPitchAndFamily;
    TCHAR lfFaceName[LF_FACESIZE];
} LOGFONT;

```

结构成员:

lfHeight —— 以逻辑单位方式指定字体的高度,如果 lfHeight 的值大于 0,则高度被转化为设备单位,并且与可用字体的单元高度相比。如果 lfHeight 等于 0,则使用默认尺寸。如果 lfHeight 小于 0,则高度被转化为设备单位,且将其绝对值与可用字体的字符高度相比较。lfHeight 的绝对值在转化后不可超过 16,384 设备单位。在所有的高度对比中,如果字体超过所要求的值,则字体映像器中大、小字体都不会超过其尺寸范围。

lfWidth —— 指定字体中字符平均宽度(用逻辑单位)。如果为 lfWidth 等于 0,则设备方向比率与可用字体的数字方向比率相比较,并寻找最佳的匹配,它由差值的绝对值决定。

lfEscapement —— 指定了偏离垂线与 x 轴在显示面上的夹角(用 0.1 度单位)。偏离垂线是从一行开始一个字符到最后一个字符的线,此角从 x 轴逆时针方向度量。

lfOrientation —— 指定字符基线和 x 轴之间的夹角(用 0.1 度单位)。此度数在坐标轴中由 x 轴逆时针方向度量时坐标系中 y 轴向下;顺时针方向从 x 轴旋转时,y 轴向上。

lfWeight —— 指定字体磅数(用每 1000 点中墨点像素数计)。lfWeight 的取值可为 0 到 1000 中的任意整数值,表 4-8 列出了常用的设置常数。

表 4-8 lfWeight 的常用设置常数

设置常数	常数数值
FW_DONTCARE	0
FW_THIN	100
FW_EXTRALIGHT	200
FW_ULTRALIGHT	200
FW_LIGHT	300
FW_NORMAL	400
FW_REGULAR	400
FW_MEDIUM	500
FW_SEMIBOLD	600
FW_DEMIBOLD	600

续表

设置常数	常数数值
FW_BOLD	700
FW_EXTRABOLD	800
FW_ULTRABOLD	800
FW_BLACK	900
FW_HEAVY	900

以上各值是大约数,实际外观依赖字体大小,有的字体仅有 FW_NORMAL,FW_REGULAR,FW_BOLD 磅数。如果 FW_DONTCARE 被指定,则使用默认磅数。

IfItalic —— 指定了字体是否为斜体。如果设置为 1 则字体为斜体,否则非斜体。

IfUnderline —— 指定字体是否有下划线。如果设置 1 则字体带下划线,否则不带下划线。

IfStrikeOut —— 指定字体字符是否突出。如果设置为非零值则突出,否则不突出。

IfCharSet —— 指定字体的字符集,其预定义的值如表 4-9 所示:

表 4-9 IfCharSet 预定义常数值

预定义常数	常数数值
ANSI_CHARSET	0
DEFAULT_CHARSETSYMBOL_CHARSET	2
SHIFTJIS_CHARSET	128
OEM_CHARSET	255

OEM 字符集与系统有关,一个系统中可能存在不同的字符集。如果应用程序使用的是未知字符集字体,则系统无法翻译或解释使用该字符集的字符串,此时应将字符串直接输入到设备驱动。字体映射器不使用 DEFAULT_CHARSET 值,应用程序可以使用此值,以使用字体名和大小来描述逻辑字体。为避免不可预料的结果,应该谨慎地使用 DEFAULT_CHARSET。

IfOutPrecision —— 指定所需的输出精度。输出精度定义输出需要的字体高度、宽度,字符方向、走格、间距之间的接近程度。当系统包含多个给定名字的字体时,应用程序可以使用 OUT_DEVICE_PRECIS、OUT_RASTER_PRECIS 和 OUT_TT_PRECIS 值控制一个字体映射器如何选择一种字体。例如,如果一个系统包含一个名叫 Symbol 的字体,以光栅和 TrueType 形式存在,指定 OUT_TT_PRECIS 使字体映射器选择 TrueType 类型(指定 OUT_TT_PRECIS 强制字体映射器选择 TrueType 字体(指定字体名与一个设备或光栅字体相匹配),即使没有同名的 TrueType 字体)。

IfClipPrecision —— 指定所需的剪切精度。剪贴精度定义了如何剪切部分超过范围的字符。如果要使用插入的只读字体,应用必须指定 CLIP_ENCAPSULATE。要创建设备旋转、TrueType 和矢量字库,应用可以用 OR 操作符将 CLIP_LH_ANGLES 值与其他 IfClipPrecision 值组合。如果 CLIP_LH_ANGLES 位被设置,所有字体的旋转依赖于坐标系的定

位都是左手方向还是右手方向。如果 CLIP_LH_ANGLES 未被设置,设置字体逆时针方向旋转,但其他字体的旋转依赖于坐标系的旋转定位。

lfQuality —— 指示字体的输出质量,定义了 GDI 逻辑字体特性和物理字体特性必须相匹配的程度。如果 lfQuality 为 DEFAULT_QUALITY 则字体的外观无关紧要。如果 lfQuality 为 DRAFT_QUALITY 则当 PROOF_QUALITY 使用时,字体的外观不太重要。对 GDI 光栅字体来说,允许缩放。黑体、斜体、下划线、突出字体和综合处理在需要时可用。如果 lfQuality 为 PROOF_QUALITY,则字体的字符质量比精确的逻辑字体特性的匹配更重要。对 GDI 光栅字体,缩放无效,大小最接近的字体被选用。黑体、斜体、下划线、突出和综合处理在需要时可用。

lfPitchAndFamily —— 指定了字体的间距和家族。其两个低位指定字体的间距,高四位指定字体家族。可用的字体家族如表 4-10 所示。

表 4-10 字体家族

字体家族常数	含义
FF_DECORATIVE	花样字体,如以前英格兰的字体
FF_DONTCARE	不明字体
FF_MODERN	笔型宽度不变的字体,有或无衬线。固定斜度的字体常是现代风格的,如 Pica, Elite 和 Courier New
FF_ROMAN	笔划宽度可变(按比例调整空间)及有衬线字体如 Times New Roman 和 Century Schoolbook。
FF_SCRIPT	与手写体相似的字体,如 Script 和 Cursive
FF_SWISS	笔划宽度可变(按比例调整空间)及不带有衬线字体,如 MS Sans Serif

可以使用布尔操作 OR 指定一个 lfPithAndFamily 值以组合一个斜体和一个家族常数。字体家族用普通方式描述了字体的外观,它们在所需铅字体无效时用于定义指定的字体。

lfFacenameCString —— 指向一个以空字符终止字符串的指针,字符串指定字体字样的名字。此字符串的长度不能长于 30 个字符,WindowsEnumFontFamilies 函数可用于枚举所有当前可用字体。如果为 NULL,则 GDI 使用一个不依赖设备的字体。

LpEnumFontFamExProc —— 指定了应用程序定义的回调函数的地址。

lParam —— 指定了 32 位应用程序定义的值,函数将此值与字体信息一起传递给回调函数。

dwFlags —— 保留参数,必须被设置为 0。

EnumFontFamiliesEx 函数并不使用标记过的字样名称来标识字符集。相反,它将正确的字体名称和独立的字符集值传递给回调函数。也就是说它只使用 LOGFONT 中的 lfCharset、lfFacename 和 lfCharset 成员。而该回调函数则负责枚举基于 LOGFONT 结构中 lfCharset 和 lfFacename 成员的字体。

如果 lfCharset 为 DEFAULT_CHARSET,并且 lfFaceName 为空字符串,则函数将为每个字符集枚举一种字体。如果 lfFaceName 非空,则函数将忽略字符集,而将枚举出指定字样的每一种字体。

如果 `lfCharset` 为有效的字符集值,并且 `lfFaceName` 为空字符串,则函数将枚举指定字符集中的每一种字体。如果 `lfFaceName` 非空,则函数枚举指定字符集和字样的每一种字体。

回调函数实际是实现字体枚举功能的主要部分。对于每个枚举字体都会调用该函数一次。字体枚举回调函数应该具有以下形式:

```
int CALLBACK EnumFontFamExProc(ENUMLOGFONTEX * lpelfe, NEWTEXTMETRICEX * lpntme, int FontType, LPARAM lParam);
```

返回值:

如果能够继续枚举,则返回非零值,否则返回零值。

参数:

`lpelfe` —— 指定了 `ENUMLOGFONTEX` 结构,其中包含了字体的逻辑属性。

`lpntme` —— 指定了字体属性,对于 TrueType 字体使用 `NEWTEXTMETRICEX` 结构,而对于其他字体则使用 `TEXTMETRIC` 结构。

`FontType` —— 指定了字体的种类,该参数的取值可以为: `DEVICE_FONTTYPE`、`RASTER_FONTTYPE` 和 `TRUETYPE_FONTTYPE`。

`lParam` —— 指定了应用程序定义的数据。

在 `CFontCombo` 类中定义了两个回调函数,分别用于枚举打印机字体和屏幕字体,其源代码如清单 4-28 和 4-29 所示:

清单 4-28 EnumFamPrinterCallbackEx() 函数

```
BOOL CALLBACK AFX_EXPORT CFontCombo::EnumFamPrinterCallbackEx(ENUMLOGFONTEX *
pelf, NEWTEXTMETRICEX * /* lpntm */, int FontType, LPVOID pThis)
{
    if (! (FontType & DEVICE_FONTTYPE))
        return 1;
    if ((FontType & TRUETYPE_FONTTYPE))
        return 1;
    DWORD dwData = PRINTER_FONT;
    ((CFontCombo *)pThis) -> AddFont(pelf -> elfLogFont.lfFaceName, dwData);
    return 1;
}
```

清单 4-29 EnumFamScreenCallbackEx() 函数

```
BOOL CALLBACK AFX_EXPORT CFontCombo::EnumFamScreenCallbackEx(ENUMLOGFONTEX *
pelf, NEWTEXTMETRICEX * /* lpntm */, int FontType, LPVOID pThis)
{
    if (FontType & RASTER_FONTTYPE)
        return 1;
    DWORD dwData;
    dwData = (FontType & TRUETYPE_FONTTYPE) ? TRUETYPE_FONT : 0;
    ((CFontCombo *)pThis) -> AddFont(pelf -> elfLogFont.lfFaceName, dwData);
    return 1;
}
```

上述函数都指定返回值为 1,表明将不断调用回调函数。在枚举打印机字体时,首先判断 `FontType` 参数是否是设备字体或 TrueType 字体,如果是则直接返回(因为不是打印机字体)。而在枚举屏幕字体时,则需要判断 `FontType` 参数是否是打印机字体(光栅字体),如果是则直接返回。如果字体符合枚举条件,则调用 `AddFont` 函数将该字体添加到组合框中。清单 4-30 所示为 `AddFont` 函数的源代码:

清单 4-30 `AddFont()` 函数

```
void CFontCombo::AddFont(CString strName, DWORD dwFlags)
{
    CFontObj * pFontObj;

    // 检查字体是否已经存在,如果不存在则添加到数组中
    if (! m_mapFonts.Lookup(strName,pFontObj))
        m_mapFonts.SetAt(strName,new CFontObj(dwFlags));
}
```

这里读者可以看到 `m_mapFonts` 数组中存放的实际上是字体的名称和类型。在组合框中只显示字体名称当然可以,但是通常字体前面都有表示字体的图标,并在当前使用字体前有分隔线。这也是软件中枚举字体的通用标准。要实现上述效果,显然应该重载 `DrawItem` 函数,清单 4-31 所示为其源代码:

清单 4-31 `DrawItem()` 函数

```
void CFontCombo::DrawItem(LPDRAWITEMSTRUCT lpDIS)
{
    ASSERT(lpDIS->CtlType == ODT_COMBOBOX);
    CDC * pDC = CDC::FromHandle(lpDIS->hDC);
    ASSERT(pDC);
    CRect rc(lpDIS->rcItem);
    // 绘制聚焦窗口
    if (lpDIS->itemState & ODS_FOCUS)
        pDC->DrawFocusRect(rc);

    int nIndexDC = pDC->SaveDC();
    CBrush brushFill;

    // 绘制选择状态
    if (lpDIS->itemState & ODS_SELECTED)
    {
        brushFill.CreateSolidBrush(::GetSysColor(COLOR_HIGHLIGHT));
        pDC->SetTextColor(::GetSysColor(COLOR_HIGHLIGHTTEXT));
    }
    else
        brushFill.CreateSolidBrush(pDC->GetBkColor());

    pDC->SetBkMode(TRANSPARENT);
    pDC->FillRect(rc, &brushFill);

    CString strCurFont, strNextFont;
    GetLBText(lpDIS->itemID, strCurFont);
```

```

CFontObj * pFontObj;
m_mapFonts.Lookup(strCurFont,pFontObj);

ASSERT(pFontObj != NULL);
DWORD dwData = pFontObj->GetFlags();

// 绘制位图
if (dwData & TRUETYPE_FONT)
    m_img.Draw(pDC,1, CPoint(rc.left,rc.top),ILD_TRANSPARENT);

if (dwData & PRINTER_FONT)
    m_img.Draw(pDC,0, CPoint(rc.left,rc.top),ILD_TRANSPARENT);

int nX = rc.left;
rc.left += GLYPH_WIDTH + 2; // 文本位置
pDC->TextOut(rc.left,rc.top,strCurFont);

// GetItemData - 返回在使用中的字体
if (GetItemData(lpDIS->itemID))
{
    GetLBText(lpDIS->itemID+1,strNextFont);
    CFontObj * pFontObjNext;
    m_mapFonts.Lookup(strNextFont,pFontObjNext);

    if (! GetItemData(lpDIS->itemID+1))
    {
        // 绘制 MRU 分隔符 =====
        TEXTMETRIC tm;
        pDC->GetTextMetrics(&tm);
        pDC->MoveTo(nX,rc.top+tm.tmHeight);
        pDC->LineTo(rc.right,rc.top+tm.tmHeight);
        pDC->MoveTo(nX,rc.top+tm.tmHeight+2);
        pDC->LineTo(rc.right,rc.top+tm.tmHeight+2);
    }
}

// 恢复设备环境
pDC->RestoreDC(nIndexDC);

```

此函数非常简单,这里就不再浪费篇幅加以解释了。到此为止,已经完成了类的主要功能设计,其他成员函数主要是完成一些辅助功能,请读者参考配套光盘 chap5/fontcombo 目录下的源代码。

另外,为了能够给用户以字体的直观信息,一般都会以相应字体绘制字体选项。要完成这个功能很简单,只要 DrawItem 函数绘制代码前使用指定字体设置设备环境即可。而此处我们将使用一种更为活泼的方式:显示提示窗口,在提示窗口中使用指定字体输出文本。为此目的,定义了一个提示窗口管理类 CTipWnd,该类的定义如清单 4-32 所示:

清单 4-32 CTipWnd 的定义

```

class CTipWnd : public CWnd
{

```



```

public:
    CTipWnd();

public:
protected:
    CFont      m_font;
    CString    m_strFont;

public:
    BOOL Create(CWnd* pParent);
    void ShowTips(CPoint pt, const CString& str);

public:
    virtual ~CTipWnd();

protected:
    //||AFX_MSG(CTipWnd)
    afx_msg BOOL OnEraseBkgnd(CDC* pDC);
    afx_msg void OnPaint();
    //||AFX_MSG
    DECLARE_MESSAGE_MAP()
};

```

在使用提示窗口前,首先调用 Create 函数创建,清单 4-33 所示为其源代码:

清单 4-33 Create()函数

```

BOOL CTipWnd::Create(CWnd* pParent)
{
    return CWnd::CreateEx(0, AfxRegisterWndClass(0), NULL,
        WS_BORDER|WS_POPUP,0,0,0,0,NULL,(HMENU)0);
}

```

该函数创建了一个具有边框和弹出式风格的窗口。创建了窗口后,就可以调用 ShowTips 函数以显示提示字体了,该函数的源代码如清单 4-34 所示:

清单 4-34 ShowTips()函数

```

void CTipWnd::ShowTips(CPoint pt,const CString& str)
{
    CSize sz;
    CDC* pDC = GetDC();

    // 如果选项发生变化,则设置字体
    if (m_strFont != str)
    {
        m_strFont = str;
        LOGFONT lf;
        ZeroMemory(&lf,sizeof(lf));
        lf.lfHeight = FONT_HEIGHT;
        strcpy(lf.lfFaceName,m_strFont);
        // 删除旧字体
        m_font.DeleteObject();
    }
}

```

```

    m_font.CreateFontIndirect(&lf);
    CFont * pFont = pDC->SelectObject(&m_font);
    //确定将显示的字符串尺寸
    sz = pDC->GetTextExtent(m_strFont);

    //设置字符串周围的空间
    sz.cx += 8;
    sz.cy += 8;

    pDC->SelectObject(pFont);
    ReleaseDC(pDC);

    SetWindowPos(0,pt.x,pt.y,sz.cx,sz.cy,SWP_SHOWWINDOW|SWP_NOACTIVATE);
    RedrawWindow(); // 立即重新绘制窗口
}
}

```

在 ShowTips 函数中,根据当前组合框中光标位置下的字体,使用 SelectObject 函数设置本窗口的输出字体。然后设置将在窗口中输出的字符串尺寸,以及窗口的位置。注意此处将窗口的属性设置为非激活态,这使得窗口不会具有输入焦点。也就是说,即使鼠标单击了工具提示窗口,焦点依然会保持在组合框所在的窗口中。

提示窗口中文本的输出是通过 OnPaint 函数完成的,其源代码如清单 4-35 所示:

清单 4-35 OnPaint()函数

```

void CTipWnd::OnPaint()
{
    CPaintDC dc(this);

    dc.SelectObject(&m_font);
    CRect rc;
    GetClientRect(rc);
    dc.DrawText(m_strFont,rc,DT_SINGLELINE|DT_VCENTER|DT_CENTER);
    // Do not call CWnd::OnPaint() for painting messages
}

```

OnPaint 是 WM_PAINT 函数的消息处理函数,每当窗口发生变化时都会产生 WM_PAINT 消息,从而触发 OnPaint 函数以重新绘制窗口。这保证了提示窗口中始终会显示提示文本。

在 CTipWnd 类中还重载了 OnEraseBkgnd 函数以绘制窗口背景(黄色背景),清单 4-36 所示为该函数的源代码:

清单 4-36 OnEraseBkgnd()函数

```

BOOL CTipWnd::OnEraseBkgnd(CDC * pDC)
{
    CBrush br(GetSysColor(COLOR_INFOBK));
    CRect rc;
    pDC->GetClipBox(rc);
    CBrush * pOldBrush = pDC->SelectObject(&br);
    pDC->PatBlt(rc.left,rc.top,rc.Width(),rc.Height(),PATCOPY);
}

```

```

    pDC->SelectObject(pOldBrush);
    return TRUE;
}

```

4.3.3 颜色选择组合框

颜色选择组合框在绘图软件中十分常见。它一般存在于软件的工具栏中。本节向读者介绍如何创建颜色选择组合框控件,如图 4-6 所示。而如何将其组合到工具栏中,会在第 7 章中进行介绍。

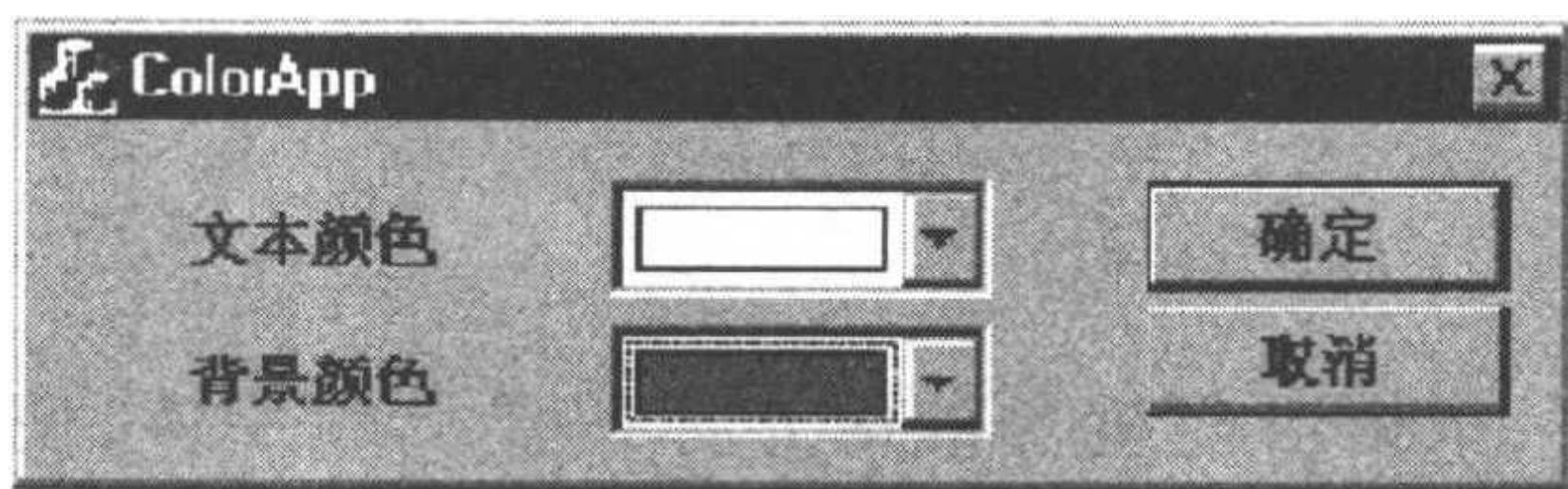


图 4-6 颜色选择组合框

首先设计颜色选择组合框的管理类 CComboColorPicker,其定义如清单 4-37 所示:

清单 4-37 CComboColorPicker 类定义

```

class CComboColorPicker : public CComboBox
{
public:
    CComboColorPicker();

private:
    bool m_bInitialized;
    static COLORREF m_rgStandardColors[];

private:
    void InitializeData();
public:
    COLORREF GetSelectedColor();
protected:
    virtual void PreSubclassWindow();

public:
    virtual ~CComboColorPicker();

protected:
    //||AFX_MSG(CComboColorPicker)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    //||AFX_MSG

    DECLARE_MESSAGE_MAP()
};

```

类中定义的 m_rgStandardColors 指定了组合框中将包括的颜色选项,用户可以在应用程序中通过直接赋值该数组以包括更多的颜色,对其的赋值如下所示:

```

COLORREF CComboColorPicker::m_rgStandardColors[] = {
    RGB(0, 0, 0),           // 黑色
    RGB(255, 255, 255),     // 白色
    RGB(128, 0, 0),         // 深红色
    RGB(0, 128, 0),         // 暗绿色
    RGB(128, 128, 0),       // 深黄色
    RGB(0, 0, 128),         // 深蓝色
    RGB(128, 0, 128),       // 深紫色
    RGB(0, 128, 128),       // 深青色
    RGB(192, 192, 192),     // 亮灰色
    RGB(128, 128, 128),     // 深灰色
    RGB(255, 0, 0),         // 红色
    RGB(0, 255, 0),         // 绿色
    RGB(255, 255, 0),       // 黄色
    RGB(0, 0, 255),         // 蓝色
    RGB(255, 0, 255),       // 紫色
    RGB(0, 255, 255),       // 青色
};

```

组合框内容的初始化是通过 OnCreate 函数完成的,其源代码如清单 4-38 所示:

清单 4-38 OnCreate()函数

```

int CComboColorPicker::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CComboBox::OnCreate(lpCreateStruct) == -1)
        return -1;
    InitializeData();
    return 0;
}

```

此外,类中还重载了 PreSubclassWindow 函数,并在其中也调用了 InitializeData 以初始化组合框内容。重载该函数主要是为了使用户能够以归类的方式使用颜色选项组合框。清单 4-39 所示为 PreSubclassWindow 函数:

清单 4-39 PreSubclassWindow()函数

```

void CComboColorPicker::PreSubclassWindow()
{
    InitializeData();
    CComboBox::PreSubclassWindow();
}

```

InitializeData 函数负责完成初始化组合框内容,其源代码如清单 4-40 所示:

清单 4-40 InitializeData()函数

```

void CComboColorPicker::InitializeData()
{
    int nItem;
    if (m_bInitialized)
        return;
}

```



```

    for (int nColor = 0; nColor < sizeof(m_rgStandardColors)/sizeof(COLORREF);
        nColor++)
    {
        nItem = AddString((LPCTSTR)m_rgStandardColors[nColor]);
        if (CB_ERRSPACE == nItem)
            break;
    }
    m_bInitialized = true;
}

```

在 `InitializeData` 函数中,将 `m_rgStandardColors` 数组中的对应元素添加到组合框中。为了保证用户在修改了 `m_rgStandardColors` 数组后,无需修改其他代码也可以直接运行;为此,在循环中使用了 `sizeof(m_rgStandardColors)/sizeof(COLORREF)` 表达式计算数组元素。当然,如果使用 MFC 就无需如此计算数组元素数,而只需要调用 `GetCount` 函数即可。

这时,读者可能也注意到了组合框中实际存储的是颜色值,而并非颜色。因此,需要调用 `DrawItem` 函数以将颜色块绘制在组合框中,其源代码如清单 4-41 所示。在函数中根据选项的颜色值设置设备环境画笔,然后使用该画笔填充选项矩形,即绘制出了如图 4-6 所示的颜色块。

清单 4-41 `DrawItem()` 函数

```

void CComboColorPicker::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
    CDC dc;
    CBrush brushBlack;
    brushBlack.CreateStockObject(BLACK_BRUSH);

    if (! dc.Attach(lpDrawItemStruct->hDC))
        return;

    COLORREF rgbTextColor = dc.GetTextColor();
    COLORREF rgbBkColor = dc.GetBkColor();

    if (lpDrawItemStruct->itemAction & ODA_FOCUS)
    {
        dc.DrawFocusRect(&lpDrawItemStruct->rcItem);
    }
    else if (lpDrawItemStruct->itemAction & ODA_DRAWENTIRE)
    {
        if (lpDrawItemStruct->itemState & ODS_FOCUS)
            dc.DrawFocusRect(&lpDrawItemStruct->rcItem);
        else
            dc.ExtTextOut(0, 0, ETO_OPAQUE, &lpDrawItemStruct->rcItem, _T(" "),
                0, NULL);
    }

    if (0 <= (int)lpDrawItemStruct->itemID)
    {
        ::InflateRect(&lpDrawItemStruct->rcItem, -2, -2);
        dc.FillSolidRect(&lpDrawItemStruct->rcItem, (COLORREF)lpDrawItem-

```

```

        Struct -> itemData);
        dc.FrameRect(&lpDrawItemStruct -> rcItem, &brushBlack);
    }

    dc.SetTextColor(rgbTextColor);
    dc.SetBkColor(rgbBkColor);
    dc.Detach();
}

```

类中还定义了 `GetSelectedColor` 函数,以完成颜色的选择,其源代码如清单 4-42 所示:

清单 4-42 `GetSelectedColor()` 函数

```

COLORREF CComboColorPicker::GetSelectedColor()
{
    int nItem = GetCurSel();
    if (CB_ERR == nItem)
        return RGB(0, 0, 0); // 设置默认选项为黑色

    return m_rgStandardColors[nItem];
}

```

到此就完成了类功能的设计,在配套光盘的 `chap5/colorpickercombo` 目录下读者可以找到一个功能更强大的颜色选择组合框类,其中实现了组合框的完全自绘制。使用按钮作为组合框中编辑框部分,而使用另外的对话框作为下拉列表框部分。但是其基本的绘制原则在这里都已经介绍过了。

4.4 增强列表框控件

`CComboBoxEx` 类是对 MFC 组合框控件类的扩展,它主要增加了对图像列表的支持,图 4-7 为其派生结构。这样,只要使用 `CComboBoxEx` 类就无需再自己定制图像绘制代码了。而只要将相应的图像添加到图像列表中即可。这时,读者可能就会想到 4.3.1 节中介绍的图标选项组合框,实际只要使用 `CComboBoxEx` 就可以轻易地实现。之所以最后才介绍 `CComboBoxEx` 类,主要是想让读者掌握一些底层的技术和技巧,这也是本书的初衷所在。

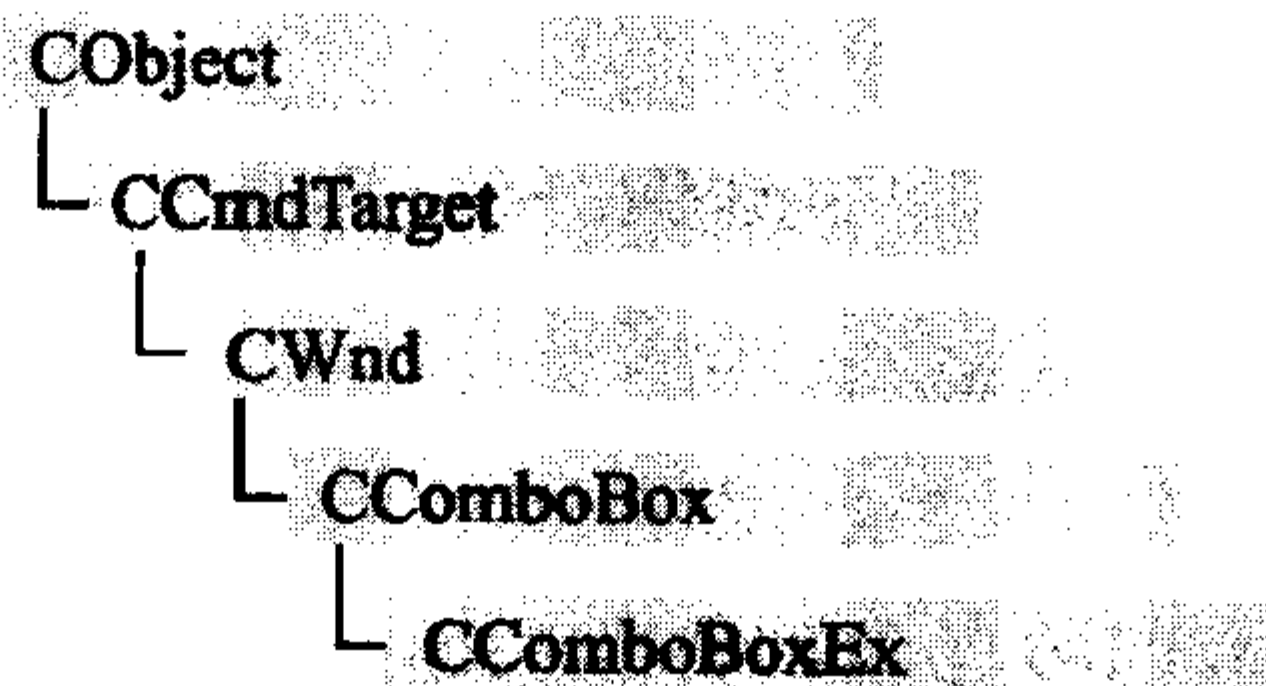


图 4-7 `CComboBoxEx` 类的派生结构

4.4.1 CComboBoxEx 类概述

在标准组合框控件中,控件的父窗口负责绘制自绘制组合框中的图像,这是通过重载 DrawItem 函数完成的。而如果使用 CComboBoxEx 类,则无需再设置 CBS_OWNERDRAW_FIXED 和 CBS_HASSTRINGS 风格以实现定制绘制。CComboBoxEx 类可以使每个选项拥有三张图像:选择态、未选态和覆盖态。

CComboBoxEx 类支持 CBS_SIMPLE、CBS_DROPDOWN、CBS_DROPDOWNLIST 和 WS_CHILD 风格。所有其他的风格常量都将被忽略。当窗口被创建后,可以调用 SetExtendStyle 来设置组合框控件的其他扩展风格,例如:设置搜索选项是否为大小写敏感、控件是否显示图像等等。

CComboBoxEx 类中增加的成员函数并不多,现介绍如下:

- CComboBoxEx

调用该函数以构造 CComboBoxEx 对象,其原型为:

```
CComboBoxEx( );
```

- Create

调用该函数以创建扩展组合框,并将其与 CComboBoxEx 相联系,其原型为:

```
BOOL Create( DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

dwStyle —— 指定了组合框控件的风格常数。其取值可以为: CBS_SIMPLE、CBS_DROPDOWN、CBS_DROPDOWNLIST 和 WS_CHILD。

rect —— 指定了组合框控件的位置和尺寸,它可以为 CRect 对象也可以为 RECT 结构。

pParentWnd —— 指定了组合框控件的父窗口,该参数不能为 NULL。

nID —— 指定了组合框控件的 ID。

创建 CComboBoxEx 对象需要两步:首先调用 CComboBoxEx 构造函数,然后在对其调用 Create 函数。当调用 Create 函数时,MFC 将初始化组合框控件。

- DeleteItem

调用该函数以从扩展组合框控件中删除一项,其原型为:

```
int DeleteItem( int iIndex );
```

返回值:

如果函数调用成功,则返回控件中剩余的选项数。如果 iIndex 参数无效,则返回 CB_ERR。

参数:

iIndex —— 指定了将被删除的选项。

• GetItem

调用该函数以得到指定条目的信息,其原型为:

```
BOOL GetItem( COMBOBOXEXITEM* pCBIItem );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

pCBIItem —— 指定了将返回选项信息的 COMBOBOXEXITEM 结构,其定义如下:

```
typedef struct {
    UINT mask;
    int iItem;
    LPTSTR pszText;
    int cchTextMax;
    int iImage;
    int iSelectedImage;
    int iOverlay;
    int iIndent;
    LPARAM lParam;
} COMBOBOXEXITEM, * PCOMBOBOXEXITEM;
```

结构成员:

mask —— 指定了将被使用的结构成员,其取值如表 4-11 所示。

表 4-11 mask 成员取值

mask 成员取值	含义
CBEIF_DI_SETITEM	控件应该储存条目值,该选项只用于 CBEN_GETDISPINFO 通告消息
CBEIF_IMAGE	必须填充 iImage 成员
CBEIF_INDENT	必须填充 iIndent 成员
CBEIF_LPARAM	必须填充 lParam 成员
CBEIF_OVERLAY	必须填充 iOverlay 成员
CBEIF_SELECTEDIMAGE	必须填充 iSelectedImage 成员
CBEIF_TEXT	必须填充 pszText 成员

iItem —— 指定了条目的索引。

pszText —— 指定了条目文本。在接受条目信息时,该成员必须被设置为将接受文本的缓冲区地址,并且将缓冲区储存长度设置为 cchTextMax 的值。如果该成员被设置为 LPSTR_TEXTCALLBACK,则控件将使用 CBEN_GETDISPINFO 通告消息得到信息。

cchTextMax —— 指定了 pszText 参数的长度,以字符为单位。如果将设置文本信息,则该成员被忽略。

iImage —— 指定了图像列表的索引。该图像将被作为条目的非选中态图像显示。如果该成员被设置为 L_IMAGECALLBACK,则控件将使用 CBEN_GETDISPINFO 通告消息得到信息。

iSelectedImage —— 指定了图像列表的索引。该图像将被作为条目的选中态图像显示。如果该成员被设置为 **L_IMAGECALLBACK**, 则控件将使用 **CBEN_GETDISPINFO** 通告消息得到信息。

iOverlay —— 指定了图像列表的索引。该图像将被作为条目的覆盖态图像显示。如果该成员被设置为 **L_IMAGECALLBACK**, 则控件将使用 **CBEN_GETDISPINFO** 通告消息得到信息。

iIndent —— 指定了条目的缩进空间, 每一个缩进值等于 10 个像素。如果该成员被设置为 **L_INDENTCALLBACK**, 则控件将使用 **CBEN_GETDISPINFO** 通告消息得到信息。

lParam —— 指定了条目的 32 位值。

- **InsertItem**

调用该函数以向扩展组合框控件中插入新选项, 其原型为:

```
int InsertItem( const COMBOBOXEXITEM* pCBItem );
```

返回值:

如果函数调用成功, 则返回新条目的索引, 否则返回 -1。

参数:

pCBItem —— 指定了将插入条目信息的 **COMBOBOXEXITEM** 结构。

- **SetItem**

调用该函数以设置扩展组合框控件中条目的属性, 其原型为:

```
BOOL SetItem( const COMBOBOXEXITEM* pCBItem );
```

返回值:

如果函数调用成功, 则返回非零值, 否则返回零值。

参数:

pCBItem —— 指定了将被设置的条目信息的 **COMBOBOXEXITEM** 结构。

- **HasEditChanged**

调用该函数以确定用户是否改变了控件编辑框中的内容, 其原型为:

```
BOOL HasEditChanged( );
```

返回值:

如果用户改变了编辑框中的内容, 则返回非零值, 否则返回零值。

- **GetExtendedStyle**

调用该函数以得到控件的扩展风格, 其原型为:

```
DWORD GetExtendedStyle( ) const;
```

返回值:

如果函数调用成功, 则返回包含控件扩展风格的 **DWORD** 值。

- **SetExtendedStyle**

调用该函数以设置控件的扩展风格, 其原型为:


```
DWORD SetExtendedStyle( DWORD dwExMask, DWORD dwExStyles );
```

返回值:

如果函数调用成功,则返回包含控件先前扩展风格的 DWORD 值。

参数:

dwExMask —— 指定了 dwExStyle 参数能够进行风格的设置。如果该参数被设置为 0, 则 dwExStyle 中指定的所有风格都将被设置。

dwExStyles —— 指定了将为控件设置的扩展风格。

- GetEditCtrl

调用该函数以得到扩展组合框控件中的编辑控件的指针,其原型为:

```
CEdit * GetEditCtrl( );
```

返回值:

如果函数调用成功,则返回 CEdit 对象指针。

- GetComboBoxCtrl

调用该函数以得到子组合框控件的指针,其原型为:

```
CComboBox * GetComboBoxCtrl( );
```

返回值:

如果函数调用成功,则返回 CComboBox 对象指针。

CComboBoxEx 控件实际是封装了 CComboBox 对象的父窗口。由该函数返回的 CComboBox 对象指针为临时对象,将在下一次空闲操作时间中被销毁。

- GetImageList

调用该函数以得到扩展组合框控件的图像列表,其原型为:

```
CImageList * GetImageList( ) const;
```

返回值:

如果函数调用成功,则返回 CImageList 对象指针,否则返回 NULL。

- SetImageList

调用该函数以设置扩展组合框控件的图像列表,其原型为:

```
CImageList * SetImageList( CImageList * pImageList );
```

返回值:

如果函数调用成功,则返回先前设置的 CImageList 对象指针。如果返回值为 NULL,则表示先前没有设置图像列表。

参数:

pImageList —— 指定了将被设置的图像列表对象指针。

4.4.2 常用操作编程

本节将介绍扩展组合框控件的常用编程方法。

1. 创建扩展组合框控件

扩展组合框控件的创建方法,与其是在对话框中使用还是在窗口中使用有关。在对话框中使用该控件时:

- (1) 在对话框编辑器中,为对话框资源添加一个组合框控件,并为其指定控件 ID。
- (2) 使用属性对话框为控件指定需要的风格。
- (3) 使用 ClassWizard 为控件添加 CComboBoxEx 类型的变量。使用该变量能够调用 CComboBoxEx 类的成员函数。
- (4) 使用 ClassWizard 为控件添加需要处理的通告消息。
- (5) 在 OnInitDialog 函数中,为控件设置附加风格。

在窗口中使用扩展组合框控件时:

- (1) 在视图或窗口类中定义控件变量。
- (2) 调用控件的 Create 函数,这一般是在 OnInitUpdate 函数中进行(它要早于对 OnCreate 的调用)。

2. 为控件设置图像列表

扩展组合框控件的最主要的特性就是能够使用图像列表。控件中的每个条目都能够显示三个不同的图像。为控件指定图像列表的一般步骤如下:

- (1) 调用 CImageList 函数以构造新的图像列表对象(也可以使用现有的图像列表对象)。
- (2) 调用 CImageList::Create 函数以初始化新的图像列表,示范代码如下:

```
m_pImageList->Create(16, 16, ILC_COLOR, 2, 2);
```

- (3) 在图像列表中添加不同状态的图像,示范代码如下:

```
m_pImageList->Add(pApp->m_hIconSelected);  
m_pImageList->Add(pApp->m_hIconNotSelected);  
m_pImageList->Add(pApp->m_hIconOverlay);
```

- (4) 调用 CComboBoxEx::SetImageList 函数将图像列表与控件相联系。

在设置了图像列表后,用户就可以为每个选项单独设置图像了。

3. 为条目设置图像

扩展组合框控件中选项使用的不同图像类型是根据 COMBOBOXEXITEM 结构的 iImage、iSelectedImage 和 iOverlay 成员值确定的。这些值是相关图像列表控件中的图像索引。默认情况下,这些成员都被设置为 0,这样控件将不为选项显示图像。如果希望为指定选项使用图像,则可以修改相应的 COMBOBOXEXITEM 结构。

在为新选项设置图像时,先使用合适的值初始化 iImage、iSelectedImage 和 iOverlay 结构成员。然后再使用初始化过的 COMBOBOXEXITEM 结构调用 CComboBoxEx::InsertItem 函数。下面给出示范代码:

```

COMBOBOXEXITEM    cbi;
CString           str;
int               nItem;

cbi.mask = CBEIF_IMAGE | CBEIF_INDENT | CBEIF_OVERLAY |
           CBEIF_SELECTEDIMAGE | CBEIF_TEXT;

cbi.iItem = i;
str.Format(_T("选项 %02d"), i);
cbi.pszText = (LPTSTR)(LPCTSTR)str;
cbi.cchTextMax = str.GetLength();
cbi.iImage = 0;
cbi.iSelectedImage = 1;
cbi.iOverlay = 2;
cbi.iIndent = (i & 0x03); //根据选项位置设置缩进

nItem = m_comboEx.InsertItem(&cbi);
ASSERT(nItem == i);

```

如果是为现存选项设置图像,则需要处理相应的 COMBOBOXEXITEM 结构,一般遵循以下步骤:

- (1) 声明一个 COMBOBOXEXITEM 结构,并设置 mask 成员为合适的值。
- (2) 以该结构为参数调用 CComboBoxEx::GetItem 函数。
- (3) 修改返回结构的 mask、iImage 和 iSelectedImage 成员为合适的值。
- (4) 以修改后的结构为参数,调用 CComboBoxEx::SetItem 函数。

为现存选项设置图像的示范代码如下:

```

COMBOBOXEXITEM    cbi;
CString           str;
int               nItem;

cbi.mask = CBEIF_IMAGE | CBEIF_SELECTEDIMAGE;
cbi.iItem = 2;
m_comboEx.GetItem(&cbi);

cbi.iImage = 1;
cbi.iSelectedImage = 0;
nItem = m_comboEx.SetItem(&cbi);
ASSERT(nItem != 0);

```

4. 处理通告消息

当用户与扩展组合框控件进行交互时,控件会向其父窗口发送通告消息。例如,当用户激活下拉列表或单击控件的编辑框时,就会向父窗口发送 CBEN_BEGINEDIT 通告。如果希望对这些通告作出处理的话,就必须定制合适的消息响应函数。常用的扩展组合框控件通告消息如表 4-12 所示。

表 4-12 扩展组合框控件通告

扩展组合框控件通告	发送情况
CBEN_BEGINEDIT	当用户激活下拉列表或单击控件的编辑框时发送
CBEN_DELETEITEM	当删除控件中的某个选项时发送
CBEN_DRAGBEGIN	当用户开始拖动显示在编辑框中的选项图像时发送
CBEN_ENDEDIT	当用户结束了对编辑框的操作,或从下拉列表中选择了条目后发送
CBEN_GETDISPINFO	当要从回调函数中得到显示信息时发送
CBEN_INSERTITEM	当向控件中插入新条目时发送

本章小结

本章主要向读者介绍了如何修改常规 Windows 组合框控件,通过本章学习读者应该达到以下几点:

- 掌握 CComboBox 类的使用。
- 掌握定制组合框控件的方法。

第 5 章 列表视图控件

列表视图控件是 Windows 应用程序中出现频率最高的控件之一,是重要的数据交互接口。本章主要向读者介绍如何设计与众不同的列表视图控件。

本章要点:

- CListCtrl 类的使用;
- 改变列表视图控件的背景;
- 改善列表视图控件的交互方式;
- 改变列表视图控件的标题显示。

5.1 列表视图控件编程基础

列表视图控件是一种常用的 Windows 控件,由 MFC 的 CListCtrl 类进行管理。CListCtrl 类的派生结构如图 5-1 所示。本节将向读者介绍其中的常用成员函数。

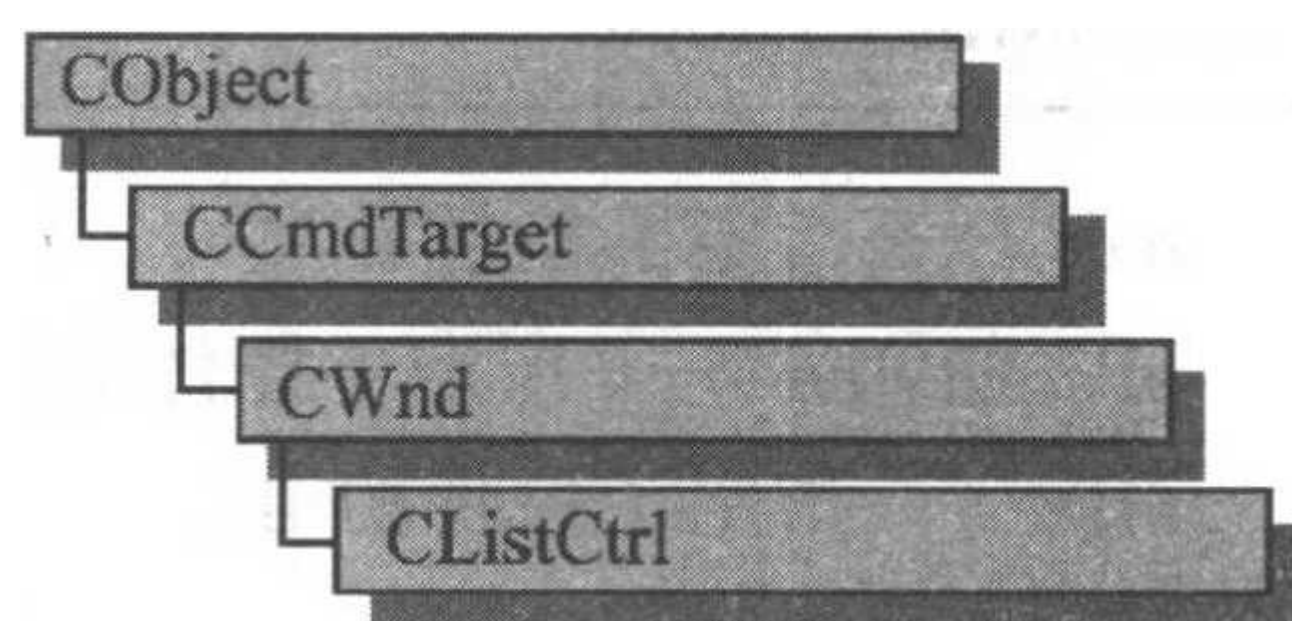


图 5-1 CListCtrl 类的派生结构

5.1.1 构造函数

CListCtrl 对象的创建函数包括 CListCtrl 和 Create,它们能够完成构造 CListCtrl 对象和创建 Windows 列表视图控件的操作。

- CListCtrl

调用该函数以构造 CListCtrl 对象,其原型为:

```
CListCtrl();
```

- Create

调用该函数以创建一个列表视图控件,其原型为:

```
BOOL Create( DWORD dwStyle, const RECT& rect, CWnd * pParentWnd, UINT nID );
```

返回值:

如果函数执行成功,则返回非零值,否则返回零值。

参数:

dwStyle ——指定了列表视图的风格,其取值可为表 5-1 中一项或几项的组合。

表 5-1 列表视图控件风格

风格常量	含义
LVS_ALIGNLEFT	在大图标和小图标视图中,条目左对齐
LVS_ALIGNTOP	在大图标和小图标视图中,条目顶对齐
LVS_AUTOARRANGE	在大图标和小图标视图中,条目自动排列
LVS_EDITLABELS	使列表视图中的条目能够被就地编辑
LVS_ICON	指定列表视图控件的显示风格为大图标视图
LVS_LIST	指定列表视图控件的显示风格为列表视图
LVS_NOCOLUMNHEADER	指定在报表视图中不显示列标题
LVS_NOLABELWRAP	指定图标视图中的条目文本为单行显示
LVS_NOSCROLL	指定列表视图控件禁止滚动
LVS NOSORTHEADER	指定列表视图控件列标题不具有按钮功能
LVS_OWNERDRAWFIXED	指定报表视图可被其父窗口绘制
LVS_REPORT	指定列表视图控件的显示风格为报表视图
LVS_SHAREIMAGELISTS	指定列表视图控件销毁时,不同时销毁图像列表
LVS_SHOWSELALWAYS	指定在列表视图控件中总是显示选择条目
LVS_SINGLESEL	指定列表视图控件中只允许单项选择
LVS_SMALLICON	指定列表视图控件的显示风格为小图标视图
LVS_SORTASCENDING	指定列表视图控件以升序排列其中条目
LVS_SORTDESCENDING	指定列表视图控件以降序排列其中条目

- rect ——指定了控件的尺寸。
- pParentWnd ——指定了控件的父窗口指针。
- nID ——指定了控件的 ID。

5.1.2 属性操作函数

CListCtrl 类的属性操作函数包括: GetBkColor、SetBkColor、GetImageList、SetImageList、GetItemCount、GetItem、SetItem、GetCallbackMask、SetCallbackMask、GetNextItem、GetFirstSelectedItemPosition、GetNextSelectedItem、GetItemRect、SetItemPosition、GetItemPosition、GetStringWith、GetEditCtrl、GetColumn、SetColumn、GetColumnWidth、SetColumnWidth、GetCheck、SetCheck、GetViewRect、GetTextColor、SetTextColor、GetTextBkColor、SetTextBkColor、GetTopIndex、GetCountPerPage、GetOrigin、SetItemState、GetItemState、GetItemText、SetItemText、SetItemCount、SetItemData、GetItemData、GetSelectedCount、GetColumnOrderArray、SetColumnOrderArray、SetIconSpacing、GetHeaderCtrl、GetHotCursor、SetHotCursor、GetSubItemRect、GetHotItem、SetHotItem、GetSelectionMark、SetSelectionMark、GetExtendedStyle、SetExtendedStyle、SubItemHitTest、GetWorkAreas、GetNumberOfWorkAreas、SetItemCountEx、SetWordItems、ApproximateViewRect、GetBkImage、SetBkImage、GetHoverImage 和 SetHoverImage,它们可以完成对列表视图控件的属性的设置和查询等操作。

- GetBkColor

调用该函数以获得列表视图控件的背景颜色,其原型为:

```
COLORREF GetBkColor( ) const;
```

返回值:

如果函数执行成功,则返回 32 位 RGB 值。

- SetBkColor

调用该函数以设置列表视图控件的背景色,其原型为:

```
BOOL SetBkColor( COLORREF cr );
```

返回值:

如果函数执行成功,则返回非零值,否则返回零值。

参数:

cr ——指定了将设置的列表视图控件背景色的 RGB 值。

- GetImageList

调用该函数以获得图像列表的指针,其原型为:

```
CImageList * GetImageList( int nImageList ) const;
```

返回值:

如果函数执行成功,则返回列表视图控件的图像列表指针。

参数:

nImageList ——指定了将获得的图像列表类型。如果 nImageList 为 LVSIL_NORMAL, 则获取大图标图像列表指针。如果 nImageList 为 LVSIL_SMALL, 则获取小图标图像列表指针。如果 nImageList 为 LVSIL_STATE, 则获取状态图像列表指针。

- SetImageList

调用该函数以将指定图像列表与列表视图控件相联系,其原型为:

```
CImageList * SetImageList( CImageList * pImageList, int nImageList );
```

返回值:

如果函数执行成功,则返回先前的图像列表指针。

参数:

pImageList ——指定了要设置的图像列表指针。

nImageList ——指定了要设置的图像列表类型。

- GetItemCount

调用该函数以获得列表视图控件中的条目数,其原型为:

```
int GetItemCount( );
```

返回值:

如果函数执行成功,则返回当前列表视图控件中的条目数。

- GetItem

调用该函数以获得列表视图控件指定条目的属性,其原型为:

```
BOOL GetItem( LVITEM* pItem ) const;
```

返回值:

如果函数执行成功,则返回真值,否则返回假值。

参数:

pItem ——指定了 LVITEM 结构指针,该结构将包含所获条目的属性。

- SetItem

调用该函数以设置列表视图控件中指定条目的属性,其原型为:

```
BOOL SetItem( const LVITEM* pItem );  
BOOL SetItem( int nItem, int nSubItem, UINT nMask, LPCTSTR lpszItem, int nImage,  
UINT nState, UINT nStateMask, LPARAM lParam );
```

返回值:

如果函数执行成功,则返回非零值,否则返回零值。

参数:

pItem ——指定了 LVITEM 结构指针,该结构包含了将设置的条目属性。

nItem ——指定了将设置的条目索引。

nSubItem ——指定了将要设置的子条目索引。

nMask ——指定了将要设置的属性。

lpszItem ——指定了条目的标签。

nImage ——指定了条目图像在图像列表中的索引。

nState ——指定了将要改变的状态值。

nStateMask ——指定了将要改变的状态。

lParam ——为与条目相联系的 32 位应用程序指定值。

- GetCallbackMask

调用该函数以得到列表视图控件的回调掩码,其原型为:

```
UINT GetCallbackMask( ) const;
```

返回值:

如果函数调用成功,则返回列表视图控件的回调掩码。

所谓“回调条目”就是列表视图中的某个条目是为应用程序,而不是为控件本身储存文本、图标或两者组合。虽然列表视图控件能够为应用程序存储这些属性,但是有时利用回调条目会更方便,例如,应用程序已经得到其中的一些信息的情况下就是如此。回调掩码指定了条目的哪个状态位是由应用程序进行维护的,以及其适用范围是整个控件还是某个条目本身。默认情况下,回调掩码为 0,表示控件将负责监控所有条目的状态。如果应用程序使用回调条目,或指定了一个非零的回调掩码,则它必须能够为列表视图控件条目提供必需的支持。

- SetCallbackMask

调用该函数以设置列表视图控件的回调掩码,其原型为:

```
BOOL SetCallbackMask( UINT nMask );
```

返回值：
如果函数调用成功,则返回非零值,否则返回零值。

参数：
nMask ——指定了回调掩码的新值。

- GetNextItem
调用该函数以在列表视图控件中寻找具有指定属性的列表条目,其原型为:

```
int GetNextItem( int nItem, int nFlags ) const;
```

返回值：
如果函数执行成功,则返回下一项目的索引,否则返回 - 1。

参数：
nItem ——指定了搜寻的起始项目。
nFlags ——指定了搜寻模式,其取值如表 5-2 所示:

表 5-2 nFlags 参数取值

nFlags 参数取值	含义
LVNL_ABOVE	搜寻指定条目之前的条目
LVNL_ALL	搜寻所有条目
LVNL_BELOW	搜寻指定条目之后的条目
LVNL_TOLEFT	搜寻指定条目左边的条目
LVNL_TORIGHT	搜寻指定条目右边的条目
LVNL_DROPHILITED	搜寻具有 LVIS_DROPHILITED 标志的条目
LVNL_FOCUSED	搜寻具有 LVIS_FOCUSED 标志的条目
LVNL_SELECTED	搜寻具有 LVIS_SELECTED 标志的条目

- GetFirstSelectedItemPosition
调用该函数以获得列表视图控件中第一个被选中的条目位置,其原型为:

```
POSITION GetFirstSelectedItemPosition( ) const;
```

如果函数执行成功,则返回条目的 POSITION 值,如果返回值为 NULL,则表示当前列表视图控件中没有条目被选。

- GetNextSelectedItem
调用该函数以获得列表视图控件中下一个被选中的列表条目,其原型为:

```
int GetNextSelectedItem( POSITION& pos ) const;
```

如果函数执行成功,则返回列表视图控件中下一个被选中的条目索引。其中参数 pos 为将接收条目 POSITION 值的变量。

- GetItemRect
调用该函数以得到当前视图中指定条目的边界矩形,其原型为:

```
BOOL GetItemRect( int nItem, LPRECT lpRect, UINT nCode ) const;
```

返回值：
如果函数调用成功，则返回非零值，否则返回零值。

参数：
nItem ——指定了条目索引。
lpRect ——指定了返回条目边界矩形的尺寸的 RECT 结构地址。
nCode ——指定了将获取的矩形类型，其取值如表 5-3 所示：

表 5-3 nCode 参数取值

nCode 参数取值	含义
LVIR_BOUNDS	返回整个条目的边界矩形，包括图标和标签
LVIR_ICON	返回图标或小图标的边界矩形
LVIR_LABEL	返回条目文本的边界矩形

• SetItemPosition

调用该函数以将列表视图控件中的指定条目移动到特定位置，其原型为：

```
BOOL SetItemPosition( int nItem, POINT pt );
```

如果函数执行成功，则返回非零值。其中参数 nItem 指定了将要设置的条目索引。
参数 pt 指定了条目的左上角位置。

• GetItemPosition

调用该函数以获得指定列表条目的位置，其原型为：

```
BOOL GetItemPosition( int nItem, LPPOINT lpPoint ) const;
```

返回值：
如果函数执行成功，则返回非零值，否则返回零值。

参数：
nItem ——指定了将要获取信息的条目索引。
lpPoint ——指定了将返回条目的左上角位置的 POINT 结构的地址。

• GetStringWidth

调用该函数以返回完全显示一个给定的字符串所需的最小列宽度，其原型为：

```
int GetStringWidth( LPCTSTR lpsz ) const;
```

返回值：
如果函数执行成功，则返回字符串 lpsz 的像素宽度。

参数：
lpsz ——为将要查询宽度的字符串。

• GetEditControl

调用该函数以得到用以编辑列表视图控件条目文本的编辑控件句柄，其原型为：


```
CEdit * GetEditControl( ) const;
```

返回值:

如果函数调用成功,则返回指向用以编辑条目文本的 CEdit 对象的指针,否则返回 NULL。

- GetColumn

调用该函数以获得列表视图控件中指定列的属性,其原型为:

```
BOOL GetColumn( int nCol, LVCOLUMN * pColumn ) const;
```

返回值:

如果函数执行成功,则返回非零值,否则返回零值。

参数:

nCol ——指定了将获取属性的列索引。

pColumn ——指定了将返回列属性为 COLDMN 结构地址。

- SetColumn

调用该函数以设置列表视图控件中指定列的属性,其原型为:

```
BOOL SetColumn( int nCol, const LVCOLUMN * pColumn );
```

返回值:

如果函数执行成功,则返回非零值,否则返回零值。

参数:

nCol ——指定了将要设置属性的列索引。

pColumn ——指定了将要设置的列属性的 COLUMN 结构地址。

- GetColumnWidth

调用该函数以得到报表视图或列表视图中指定列的宽度,其原型为:

```
int GetColumnWidth( int nCol ) const;
```

返回值:

如果函数调用成功,则返回指定列的像素宽度。

参数:

nCol ——指定了将被获取宽度的列索引。

- SetColumnWidth

调用该函数以设置报表视图或列表视图中指定列的宽度,其原型为:

```
BOOL SetColumnWidth( int nCol, int cx );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

nCol ——指定了将被设置宽度的列索引。在列表视图中,该参数必须为 -1。

cx ——指定了列的新宽度,其取值可以为 LVSCW_AUTOSIZE 或 LVSCW_AUTOSIZE_USE_HEADER。

- GetCheck

调用该函数以得到指定条目的当前选择状态,其原型为:

```
BOOL GetCheck( int nItem ) const;
```

返回值:

如果条目被选择,则返回非零值,否则返回零值。

参数:

nItem ——指定了被查询状态的条目索引。

- SetCheck

调用该函数以设置指定条目的选择状态,其原型为:

```
BOOL SetCheck( int nItem, BOOL fCheck = TRUE );
```

返回值:

如果条目被选择,则返回非零值,否则返回零值。

参数:

nItem ——指定了将被设置状态的条目索引。

fCheck ——指定了条目的选择状态(状态图像的显示与否)。默认情况下,fCheck 为 TRUE,即状态图像可见,否则状态图像不可见。

- GetViewRect

调用该函数以获得列表视图控件中所有条目的边界矩形,其原型为:

```
BOOL GetViewRect( LPRECT lpRect ) const;
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

lpRect ——指定了用以接收矩形尺寸的 RECT 结构地址。

要调用 GetViewRect 函数,列表视图控件必须是图标视图或小图标视图类型。

- GetTextColor

调用该函数以获得列表视图控件的文本颜色,其原型为:

```
COLORREF GetTextColor( ) const;
```

返回值:

如果函数执行成功,则返回文本颜色的 RGB 值。

- GetTextBkColor

调用该函数以得到列表视图控件中文本背景色,其原型为:

```
COLORREF GetTextBkColor( ) const;
```

返回值:

如果函数调用成功,则返回文本背景颜色的 RGB 值。

- GetTopIndex

调用该函数以得到列表视图控件(列表视图或报表视图类型)的最顶部可见条目索引。

引,其原型为:

```
int GetTopIndex( ) const;
```

返回值:

如果函数调用成功,则返回控件最顶部可见条目的索引。

- GetCountPerPage

调用该函数以得到列表视图控件(列表视图或报表视图类型)中可见区域所能包含的条目数,其原型为:

```
int GetCountPerPage( ) const;
```

返回值:

如果函数调用成功,则返回控件可见区域所能包含的条目数。

- GetOrigin

调用该函数以得到当前列表视图控件的原点,其原型为:

```
BOOL GetOrigin( LPPOINT lpPoint ) const;
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。然而,如果控件为报表视图,则返回值总是为零。

参数:

lpPoint ——指定将接收返回的圆点坐标的 POINT 结构地址。

- SetItemState

调用该函数以改变列表视图控件中指定条目的状态,其原型为:

```
BOOL SetItemState( int nItem, LVITEM* pItem );  
BOOL SetItemState( int nItem, UINT nState, UINT nMask );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

nItem ——指定了将被设置状态的条目索引。

pItem ——指定了包含将设置的条目信息的 LVITEM 结构地址。结构中的 stateMask 成员指定了将改变的状态位,而其 state 成员则包含这些位的新值,结构的其他成员将被忽略。

nState ——指定了状态位的新值。

nMask ——指定列将改变的状态位。

LVITEM 结构的定义如下:

```
typedef struct _LVITEM {  
    UINT mask;  
    int iItem;  
    int iSubItem;
```

```
    UINT state;
    UINT stateMask;
    LPTSTR pszText;
    int cchTextMax;
    int iImage;
    LPARAM lParam;
    # if (_WIN32_IE >= 0x0300)
        int iIndent;
    # endif
} LV_ITEM;
```

结构成员：
mask ——指定了结构中将使用的成员，其取值如表 5-4 所示。

表 5-4 mask 成员取值

mask 成员取值	含义
LVIF_IMAGE	表示成员 iImage 有效
LVIF_NORECOMPUTE	表示当接收到 LVM_GETIMAGE 消息时，控件不会生成 LVN_GETDISPINFO 以获得文本信息。否则，pszText 成员将包括 LPSTR_TEXTCALLBACK
LVIF_INDENT	表示成员 iIndent 成员有效
LVIF_PARAM	表示成员 lParam 有效
LVIF_DL_SETITEM	操作系统应该存储被查询的列表条目信息，以便不必再次查询。该标志只与 LVN_GETDISPINFO 通告消息共同使用
LVIF_STATE	表示成员 state 有效
LVIF_TEXT	表示成员 pszText 有效

iItem ——指定了条目索引。
iSubItem ——指定了条目的子项索引。
state ——指定了条目的状态、状态图像以及覆盖图像。而 stateMask 则指定了该成员的有效位。该成员的 0 到 7 位包含了条目的状态标志，其取值如表 5-5 所示。

表 5-5 state 成员取值

state 成员取值	含义
LVIS_CUT	表示条目被选中用作剪贴
LVIS_DROPHILITED	表示条目为高亮显示的拖动对象
LVIS_ACTIVATING	表示条目被 LVN_ITEMACTIVATE 通告激活
LVIS_FOUSED	表示条目获得焦点
LVIS_SELECTED	表示条目被选中
LVIF_TEXT	表示成员 pszText 有效

成员的 8 到 11 位指定列的覆盖图像索引(以 0 为基)。无论是原始尺寸的图标图像列表，或是小图标图像列表都有覆盖图像。覆盖图像将叠加到条目图标图像之上。如果这些位为 0，则条目没有覆盖图像。使用 LVIS_OVRLAYMASK 掩码，可以将这些位分离出

来。要在此成员中设置覆盖图像索引,则应该使用 INDEXTOOVERLAYMASK 宏。图像列表的覆盖图像由 ImageList::SetOverlayImage 函数设置。

成员的 12 到 15 位为指定列状态图像索引。状态图像显示在条目图标旁边,以标识应用程序定义的状态。如果这些位为 0,则条目没有状态图像。使用 LVIS_STATEIMAGE MASK 掩码,可以将这些位分离出来。使用 INDEXTOSTATEIMAGE MASK 宏,能够设置该成员的状态图像索引。状态图像索引指定了状态图像列表中的图像位置,而该图像列表是通过 LVM_SETIMAGELIST 消息指定的。

statMask ——指定了将获取或设置状态成员的哪一位,其可能的取值与 state 相同。例如,将该成员设置为 LVIS_SELECTED,则标识将获取条目的选择状态。

该成员允许用户无需得到所有的状态,就能够修改其中的一个或多个。例如,将该成员设置为 LVIS_SELECTED,而将 state 设置为 0,则将使相应条目的选择状态被清除,而该条目的其他状态则不受影响。要获取或修改所有状态,则将该成员设置为 (UINT) - 1。使用 ListView::SetItemState 函数能够设置或清除这些位。

pszText ——如果结构指定了条目属性,则该成员指定了包含条目文本的缓冲区地址。如果该成员为 LPSTR_TEXTCALLBACK,则条目为回调条目。如果列表视图控件具有 LVS_SORTASCENDING 或 LVS_SORTDESCENDING 风格,则不要设置该成员。如果结构将接收条目属性,则该成员该将接收条目文本。

cchTextMax ——指定了 pszText 成员的长度,如果结构指定了条目属性,则该成员将被忽略。

iImage ——指定了条目图标在控件图像列表中的索引,该成员可以用于大图标或小图标图像列表。

如果该成员为 L_IMAGECALLBACK,则父窗口负责存储索引。在这种情况下,当需要显示图像时,列表视图控件向父窗口发送 LVN_GETDISPINFO 通告消息,以得到其索引。

lParam ——为条目的 32 位值。如果使用 LVM_SORTITEMS 消息,则列表视图控件将该值发送给应用程序定义的比较函数。也可以使用 LVM_FINDITEM 消息来得到列表视图控件中符合 lParam 值的条目。

条目的状态标识着条目是否可用,用户的动作可能影响条目的状态。列表视图控件能够通过改变状态位(而无需用户直接操作列表视图)来改变条目的状态。例如,用户在编辑框中输入某个条目的名称,应用程序在列表视图控件中搜索并定位该条目,然后改变该条目的状态,使其处于选中状态。

- GetItemState

调用该函数以得到指定列表视图控件条目的状态,其原型为:

```
UINT GetItemState( int nItem, UINT nMask ) const;
```

返回值:

如果函数调用成功,则返回指定条目的状态标志。

参数:

nItem ——指定将被获取状态的条目索引。

nMask ——将返回条目的状态标志。

- SetItemText

调用该函数以改变条目或条目子项的文本,其原型为:

```
BOOL SetItemText( int nItem, int nSubItem, LPTSTR lpszText );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

nItem ——指定将被设置的条目索引。

nSubItem ——指定了将被设置文本的条目子项索引。

lpszText ——指定了将设置的新条目文本。

- GetItemText

调用该函数以得到列表视图控件中的指定条目文本,其原型为:

```
int GetItemText( int nItem, int nSubItem, LPTSTR lpszText, int nLen ) const;  
CString GetItemText( int nItem, int nSubItem ) const;
```

返回值:

对于 int 版本函数,其返回值为所指定的文本长度;而 CString 版本的函数,其返回值为包含条目文本的 CString 对象。

参数:

nItem ——指定了将被获取文本的条目索引。

nSubItem ——指定了将被获取文本的条目子项索引。

lpszText ——将返回条目文本的缓冲区指针。

nLen ——指定了 lpszText 缓冲区的长度。

如果 nSubItem 为 0,则函数将获取条目的标签。如果 nSubItem 不为 0,则将获取指定条目子项的文本。

- SetItemData

调用该函数以设置与条目相关的 32 位应用程序指定值,其原型为:

```
BOOL SetItemData( int nItem, DWORD dwData );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

nItem ——指定了将被设置数据的条目索引。

dwData ——指定了将设置的 32 位数据。

被设置的值实际上就是 LVITEM 结构中的 lParam 成员。

- GetItemData

调用该函数以得到与指定条目相关的 32 位应用程序指定值,其原型为:

```
DWORD GetItemData( int nItem ) const;
```

返回值:

如果函数调用成功,则返回 32 位应用程序指定值。

参数:

nItem ——指定了将得到其数据的条目索引。

- GetSelectedCount

调用该函数以得到列表视图控件中被选中的条目数,其原型为:

```
UINT GetSelectedCount( ) const;
```

返回值:

如果函数调用成功,则返回列表视图控件中被选中的条目数。

- SetColumnOrderArray

调用该函数以设置列表视图控件中列的顺序,其原型为:

```
BOOL SetColumnOrderArray( int iCount, LPINT piArray );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

piArray ——指定了缓冲区指针,其中包含列表视图控件中的列索引值(从左到右)。缓冲区必须足够大以包含列表视图控件中的所有列。

iCount ——指定了列表视图控件的列数。

- GetColumnOrderArray

调用该函数以得到列表视图控件的列顺序,其原型为:

```
BOOL GetColumnOrderArray( LPINT piArray, int iCount = -1 );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

piArray ——将返回列表视图控件的列顺序。缓冲区必须足够大以包含列表视图控件的所有列。

iCount ——指定了列表视图控件的列数。如果该参数为 -1,则列数将自动由框架获得。

- SetIconSpacing

调用该函数以设置图标的间距,其原型为:

```
CSize SetIconSpacing( int cx, int cy );  
CSize SetIconSpacing( CSize size );
```

返回值:

如果函数调用成功,则返回先前图标间距。

参数:

cx ——指定了图标的 x 间距。

cy ——指定了图标的 y 间距。

size ——指定了包含图标 x 和 y 间距的 CSize 对象。

- **GetHeaderCtrl**

调用该函数以得到列表视图控件的标头控件指针,其原型为:

```
CHeaderCtrl * GetHeaderCtrl( );
```

返回值:

如果函数调用成功,则返回列表视图控件所使用的标头控件指针。

- **GetHotCursor**

调用该函数以得到列表视图控件的热追踪(hot tracking)光标,其原型为:

```
HCURSOR GetHotCursor( )
```

返回值:

如果函数调用成功,则返回当前列表视图控件所用的热光标资源句柄。

热光标只是当使用滞留选择时才会使用。所谓滞留选择,就是用户无需单击条目,而只是将鼠标停留在条目上就会导致该条目被选择。这种特性可以通过设置扩展风格 LVS_EX_TRACKSELECT 得到。

- **SetHotCursor**

调用该函数以设置列表视图控件热追踪所用的光标,其原型为:

```
HCURSOR SetHotCursor( HCURSOR hc );
```

返回值:

如果函数调用成功,则返回先前列表视图控件所用的热光标资源句柄。

参数:

hc ——指定了将设置的光标资源句柄。

- **GetSubItemRect**

调用该函数以得到列表视图控件中条目的子项的边界矩形,其原型为:

```
BOOL GetSubItemRect( int iItem, int iSubItem, int nArea, CRect& ref );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

iItem ——指定了条目索引。

iSubItem ——指定了条目的子项索引。

nArea ——指定将获取的边界矩形类型,其可取值包括:LVIR_BOUNDS、LVIR_ICON、LVIR_LABEL(含义参见表 3-3)。

ref ——将返回条目的子项的边界矩形。

- GetHotItem

调用该函数以得到列表视图控件中当前光标下的条目(热条目),其原型为:

```
int GetHotItem( );
```

返回值:

如果函数调用成功,则返回列表视图控件的当前热条目。

热条目的概念也只能用于热追踪模式。

- SetHotItem

调用该函数以设置当前列表视图控件中的热条目,其原型为:

```
int SetHotItem( int iIndex );
```

返回值:

如果函数调用成功,则返回先前热条目的索引。

参数:

iIndex ——指定了将被设置为热条目的索引。

- GetSelectionMark

调用该函数以得到列表视图控件的选择标记,其原型为:

```
int GetSelectionMark( );
```

返回值:

如果函数调用成功,则返回选择标记(从0开始计算);如果返回-1,则表示当前没有选择标记。

- SetSelectionMark

调用该函数以设置列表视图控件的选择标记,其原型为:

```
int SetSelectionMark( int iIndex );
```

返回值:

如果函数调用成功,则返回先前选择标记(从0开始计算);如果返回-1,则表示当前没有选择标记。

参数:

iIndex ——指定了多重选项的第一个条目索引。

- GetExtendedStyle

调用该函数以得到列表视图控件的扩展风格,其原型为:

```
DWORD GetExtendedStyle( );
```

返回值:

如果函数调用成功,则返回列表视图控件所使用的扩展风格。列表视图控件能够使用的扩展风格如表5-6所示。

表 5-6 列表视图控件扩展风格

扩展风格	含义
LVS_EX_CHECKBOXES	使列表视图控件的条目前出现复选框
LVS_EX_FLATSB	使列表视图控件的滚动条为平滑滚动条
LVS_EX_FULLROWSELECT	对列表视图控件中条目进行整行选择,即选择某条目时,该条目的所在行都被选择。该选项只对 LVS_REPORT(报表)风格有效
LVS_EX_GRIDLINES	将在列表视图控件中绘制网格线,该选项只对 LVS_REPORT 风格有效
LVS_EX_HEADERDRAGDROP	将在列表视图控件中允许列的拖拽操作,该风格只对 LVS_REPORT 风格有效
LVS_EX_INFOTIP	列表视图控件将向父窗口发送 LVN_GETINFOTIP 消息 ,以显示工具提示。该选项只对 LVS_ICON 风格有效
LVS_EX_MULTIWORKAREAS	如果列表视图控件具有 LVS_AUTOARRANGE 风格,则除非定义了工作区域,否则列表视图控件不对图标进行自动排布
LVS_EX_ONECLICKACTIVATE	当用户单击某条目时,列表视图控件将向父窗口发送 LVN_ITEMACTIVATE 消息
LVS_EX_REGIONAL	列表视图控件将创建一个包含条目图标和文本的区域,而且将该窗口区域设置为正在使用的窗口区域。该选项只对 LVS_ICON 风格有效
LVS_EX_SUBITEMIMAGES	允许列表控制视图的子条目显示图标。该选项只对 LVS_REPORT 风格有效
LVS_EX_TRACKSELECT	允许列表视图控件进行跟踪选择,即鼠标在某一条目上停留一段时间后,该条目被自动选择。停留时间的设定可通过 SetHoverTime 成员函数完成
LVS_EX_TWOCLICKACTIVATE	列表视图控件中的条目被双击时才被激活
LVS_EX_UNDERLINECOLD	使列表视图控件中的未被选择的条目文本下出现下划线
LVS_EX_UNDERLINEHOT	使列表视图控件中的被选条目文本下出现下划线

下面的示范代码使列表视图控件具有整行选择、跟踪选择等风格：

```
m_list.SetExtendedStyle(LVS_EX_HEADERDRAGDROP|LVS_EX_FULLROWSELECT|
    LVS_EX_GRIDLINES|LVS_EX_TRACKSELECT);
```

• SetExtendedStyle

调用该函数以设置列表视图控件的扩展风格,其原型为：

```
DWORD SetExtendedStyle( DWORD dwNewStyle );
```

返回值：

如果函数调用成功,则返回列表视图控件先前所使用的扩展风格。

参数：

dwNewStyle ——指定了列表视图控件将使用的扩展风格,其取值参见表 6-6。

• SubItemHitTest

调用该函数以确定指定位置处的列表视图条目,其原型为：


```
int SubItemHitTest( LPLVHITTESTINFO pInfo );
```

返回值:

如果函数调用成功,则返回条目或条目的索引,否则返回 -1。

参数:

pInfo —— 指定了 LVHITTESTINFO 结构,其中包含单击测试的信息。它用于与 LVM_HITTEST 和 LVM_SUBITEMHITTEST 等相关的宏。该结构取代了原先的 LV_HITTESTINFO 结构。LVHITTESTINFO 结构的定义如下:

```
typedef struct _LVHITTESTINFO {
    POINT pt;
    UINT flags;
    int iItem;
    int iSubItem;
} LVHITTESTINFO, FAR * LPLVHITTESTINFO;
```

结构成员:

pt —— 指定了单击的位置(客户区坐标)。

flags —— 指定了用于接收单击测试信息结果的变量,该成员的取值如表 5-7 所示:

表 5-7 flags 成员取值

flags 成员取值	含义
LVHT_ABOVE	单击位置在列表视图控件客户区之上
LVHT_BELOW	单击位置在列表视图控件客户区之下
LVHT_NOWHERE	单击位置在列表视图控件客户窗口中,但是并不在条目上
LVHT_ONITEMICON	单击位置在条目的图标上
LVHT_ONITEMLABEL	单击位置在条目文本上
LVHT_ONITEMSTATEICON	单击位置在条目的状态图标上
LVHT_TOLEFT	单击位置在列表视图控件客户区左边
LVHT_TORIGHT	单击位置在列表视图控件客户区右边

可以通过测试 LVHT_ABOVE、LVHT_BELOW、LVHT_TOLEFT 和 LVHT_TORIGHT 来确定是否要滚动列表视图控件。这四个值可以两两组合使用,例如如果位置在客户区的左上角,则应该设置 LVHT_ABOVE 和 LVHT_TOLEFT。

可以检查 LVHT_ONITEM 以确定指定位置是否在某个条目之上。该值可以通过 OR 操作符与 LVHT_ONITEMICON、LVHT_ONITEMLABEL 和 LVHT_ONITEMSTATEICON 组合使用。

iItem —— 将返回符合条件的条目索引。如果是条目的子项,则该值为包含子项的条目索引。

iSubItem —— 将返回符合条件的条目的子项。当测试的是条目时,该成员为 0。

- GetWorkAreas

调用该函数以得到列表视图控件的当前工作区域,其原型为:

```
void GetWorkAreas( int nWorkAreas, LPRECT prc ) const;
```

参数:

nWorkAreas ——指定了 prc 数组包含的 RECT 结构数。

prc ——指定了一个 RECT 或 CRect 对象的数组,它用于返回列表视图控件的工作区域。这些结构返回客户区坐标。

- GetNumberOfWorkAreas

调用该函数以得到列表视图控件的当前工作区域数,其原型为:

```
UINT GetNumberOfWorkAreas( ) const;
```

返回值:

目前尚未使用。

- SetItemCountEx

调用该函数以设置虚列表视图控件的条目数,其原型为:

```
void SetItemCountEx( int iCount, DWORD dwFlags = LVSI CF_NOINVALIDATEALL );
```

参数:

nItems ——指定了列表视图控件最多能包含的条目数。

dwFlags ——指定了列表视图控件在重新设置其条目数后的行为。如果该参数的取值为 LVSI CF_NOINVALIDATEALL,则表示除非当前显示的条目受到影响,否则列表视图控件将不会被重新绘制(默认值)。如果该参数的取值为 LVSI CF_NOSCROLL,则列表视图控件将在其条目数改变后重新绘制。

- SetWorkAreas

调用该函数以设置列表视图控件中能够显示图标的区域,其原型为:

```
void SetWorkAreas( int nWorkAreas, LPRECT lpRect );
```

参数:

nWorkAreas ——指定了 lpRect 数组中包含的 RECT 结构或 CRect 对象数目。

lpRect ——指定了一个 RECT 结构或 CRect 对象数组,其中包括了新的列表视图控件工作区域。这些区域必须使用客户区坐标设置。如果该参数为 NULL,则工作区被设置为控件的客户区域。

- ApproximateViewRect

调用该函数以确定列表视图控件中显示条目所需的高度和宽度,其原型为:

```
CSize ApproximateViewRect( CSize sz = CSize( -1, -1), int iCount = -1 ) const;
```

返回值:

如果函数调用成功,则返回包含显示条目所需的高度和宽度(像素为单位)的 CSize 对象。

参数:

sz ——指定了包含预期尺寸的 CSize 对象。如果没有指定尺寸,则框架将使用控件当前的宽度和高度值。

iCount ——指定了控件显示的条目数。如果该参数为 -1, 则框架使用当前控件中的总条目设置该参数。

- GetBkImage

调用该函数以得到列表视图控件的背景图像, 其原型为:

```
BOOL GetBkImage( LVBKIMAGE * plvbkImage ) const;
```

返回值:

如果函数调用成功, 则返回非零值, 否则返回零值。

参数:

plvbkImage ——为 LVBKIMAGE 指针, 将返回列表视图控件的背景图像。

- SetBkImage

调用该函数以设置列表视图控件的背景图像, 其原型为:

```
BOOL SetBkImage( LVBKIMAGE * plvbkImage );  
BOOL SetBkImage( HBITMAP hbm, BOOL fTile = TRUE, int xOffsetPercent = 0, int  
yOffsetPercent = 0 );  
BOOL SetBkImage( LPTSTR pszUrl, BOOL fTile = TRUE, int xOffsetPercent = 0, int  
yOffsetPercent = 0 );
```

返回值:

如果函数调用成功, 则返回非零值, 否则返回零值。

参数:

plvbkImage ——指定了将用以设置背景图像的 LVBKIMAGE 结构指针。

hbm ——指定了位图句柄。

pszUrl ——指定了包含背景图像的 URL 字符串指针。

fTile ——指定了背景图像的显示方式。如果该参数为非零值, 则图像平铺显示。

xOffsetPercent ——指定了背景图像左上角与列表视图控件原点间的 x 距离。

yOffsetPercent ——指定了背景图像左上角与列表视图控件原点间的 y 距离。

由于 SetBkImage 函数使用了 OLE COM 功能, 因此在调用 SetBkImage 函数之前, 必须首先初始化 OLE 库。最好是在应用程序启动时初始化 OLE 库, 而在应用程序终止时关闭 OLE 库。如果使用 ActiveX 技术, 则这个工作是由 MFC 应用程序自动完成的。

- GetHoverTime

调用该函数以得到列表视图控件的热追踪时间, 其原型为:

```
DWORD GetHoverTime( ) const;
```

返回值:

如果函数调用成功, 则返回条目被选中的热追踪时间, 也就是鼠标在条目上的滞留时间(以毫秒为单位)。如果返回值为 -1, 则热追踪时间为默认值。

- SetHoverTime

调用该函数以设置列表视图控件的热追踪时间, 其原型为:

```
DWORD SetHoverTime( DWORD dwHoverTime = (DWORD) - 1 );
```

返回值:

如果函数调用成功,则返回先前的热追踪时间。

参数:

dwHoverTime ——指定了新的热追踪时间。

- **SetTextColor**

调用该函数以设置列表视图控件的文本颜色,其原型为:

```
BOOL SetTextColor( COLORREF cr );
```

返回值:

如果函数执行成功,则返回非零值。

参数:

cr ——指定了将要设置的文本颜色 RGB 值。

- **SetItemCount**

调用该函数以设置列表视图控件可以包含的条目数目,其原型为:

```
void SetItemCount( int iCount );
```

参数:

iCount ——指定了列表视图控件能够包含的条目数。

5.1.3 常规操作函数

CListCtrl 类的常规操作函数包括: InsertItem、DeleteItem、DeleteAllItems、FindItem、SortItems、HitTest、EnsureVisible、Scroll、RedrawItems、Update、Arrange、EditLabel、InsertColumn、DeleteColumn 和 CreateDragImage,它们可以完成向列表视图控件中插入新条目或新列等操作。

- **InsertItem**

调用该函数以向列表视图控件中插入新条目,其原型为:

```
int InsertItem( const LVITEM* pItem );
```

返回值:

如果函数执行成功,则返回新条目索引值。

参数:

pItem ——为 LVITEM 结构,其中包含了将插入的条目信息。

- **DeleteItem**

调用该函数以删除列表视图控件中的指定条目,其原型为:

```
BOOL DeleteItem( int nItem );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

nItem ——指定了将被删除的条目索引。

- DeleteAllItems

调用该函数以删除列表控件中的所有条目,其原型为:

```
BOOL DeleteAllItems( );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

- FindItem

调用该函数以在列表视图控件中检索条目,其原型为:

```
int FindItem( LVFINDINFO * pFindInfo, int nStart = -1 ) const;
```

返回值:

如果函数调用成功,则返回条目索引,否则返回 -1。

参数:

pFindInfo ——指定了包含所检索条目信息的 LVFINDINFO 结构。

nStart ——指定了开始检索操作的条目索引。如果该参数为 -1,则从开头开始检索。

如果 nStart 不为 -1,则其所指定的条目是不包括在检索操作中的。

LVFINDINFO 结构的定义如下

```
typedef struct tagLVFINDINFO
{
    UINT flags;
    LPCTSTR psz;
    LPARAM lParam;
    POINT pt;
    UINT vkDirection;
} LVFINDINFO, FAR * LPFINDINFO;
```

结构成员:

flags ——指定了检索操作的类型,其取值如表 5-8 所示。

表 5-8 flags 成员取值

flags 成员取值	含义
LVFL_PARAM	检索是基于 lParam 成员的,亦即符合条件的条目的 LVITEM 结构的 lParam 成员,与本结构中 lParam 成员必须一致。如果指定了该值,则所有其他值都被忽略
LVFL_PARTIAL	检查条目文本是否以 psz 成员指定的文本为起始。这时需要指定 LVFL_STRING 标志

续表

flags 成员取值	含义
LVFL_STRING	检索是基于条目文本的。除非指定了附加值,否则符合条件的条目文本必须与 psz 成员严格一致
LVFL_WRAP	如果没有找到符合条件的条目,则从开头继续检索
LVFL_NEARESTXY	检索最靠近 pt 成员的条目,检索方向由 vkDirection 成员指定

psz ——指定了将用于检索的文本,这时必须指定 LVFL_STRING 或 LVFL_PARTIAL 标志。

lParam ——指定了用于与 LVITEM 结构的 lParam 成员比较的值。此时 flags 成员必须为 LVFL_PARAM。

pt ——指定了检索的起始位置,只有当 flags 为 LVFL_NEARESTXY 时,才使用该成员。

vkDirection ——指定了检索的方向,其中包含了箭头建的虚键码。该成员只在 flags 为 LVFL_NEARESTXY 时使用。

- SortItems

调用该函数以排序列表视图控件中的条目,其原型为:

```
BOOL SortItems( PFNLVCOMPARE pfnCompare, DWORD dwData );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

pfnCompare ——指定了应用程序定义的比较函数的地址。在排序操作的每一次比较中,都会调用比较函数。这个函数必须为类的静态成员,或为不属于任何类的独立函数。

dwData ——指定了将传递给比较函数的应用程序定义值。

列表视图中的条目的索引会发生相应变化,以体现排序的结果。比较函数具有以下形式:

```
int CALLBACK CompareFunc(LPARAM lParam1, LPARAM lParam2, LPARAM lParamSort);
```

如果第一个条目在第二个条目之前,则比较函数必须返回负值;而如果第一个条目在第二个条目之后,则比较函数必须返回正值;如果两个条目相同,则比较函数应该返回 0。其中 lParam1 和 lParam2 参数分别指定了将进行比较的条目数据。而 lParamSort 参数则与 dwData 相等。

- HitTest

调用该函数以确定列表视图控件的指定位置处是否存在条目,其原型为:

```
int HitTest( LVHITTESTINFO * pHitTestInfo ) const;  
int HitTest( CPoint pt, UINT * pFlags = NULL ) const;
```

返回值:

如果函数调用成功,则返回由 pHitTestInfo 指定位置处的条目索引。如果找不到符合条件的条目,则返回 -1。

参数:

pHitTestInfo ——指定了包含单击测试信息的 LVHITTESTINFO 结构指针。

pt ——指定用于测试的点。

pFlags ——指定了用于返回测试结果的变量指针。

该函数的参数和作用与 SubItemHitTest 类似,请读者比较阅读。

- EnsureVisible

调用该函数以确保指定的列表视图条目可见,其原型为:

```
BOOL EnsureVisible( int nItem, BOOL bPartialOK );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

nItem ——指定了必须可见的条目索引。

bPartialOK ——指定了是否允许部分可见。

列表视图控件将在必要时进行滚动,以确保指定条目可见。如果 bPartialOK 参数为非零值,则当条目部分可见时,列表视图控件不进行滚动。

- Scroll

调用该函数以滚动列表视图控件,其原型为:

```
BOOL Scroll( CSize size );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

size ——指定了包含水平和垂直滚动尺寸的 CSize 对象。其中 CSize 对象的 y 成员需要除以列表视图控件的行高,从而得到需要滚动的行数,然后控件据此进行滚动。

- RedrawItems

调用该函数以重新绘制指定范围中的条目,其原型为:

```
BOOL RedrawItems( int nFirst, int nLast );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

nFirst ——指定了将被重新绘制的第一个条目索引。

nLast ——指定了将被重新绘制的最后一个条目索引。

指定的条目实际上直到接收到 WM_PAINT 消息后,才会被真正重新绘制。如果需要立即重新绘制,则可以在调用 RedrawItems 后,立即调用 UpdateWindows 函数。

- Update

调用该函数以强制列表视图控件重新绘制指定条目,其原型为:

```
BOOL Update( int nItem );
```

返回值：
如果函数调用成功，则返回非零值，否则返回零值。

参数：
nItem ——指定了将被重新绘制的条目索引。
如果列表视图控件具有 LVS_AUTOARRANGE 风格，该函数还会重新排列控件中的条目。

• Arrange
调用该函数以在图标型列表视图控件中，重新排列条目，其原型为：

```
BOOL Arrange( UINT nCode );
```

返回值：
如果函数调用成功，则返回非零值，否则返回零值。

参数：
nCode ——指定了条目排列的类型，其取值如表 5-9 所示。

表 5-9 nCode 参数取值

nCode 参数取值	含义
LVA_ALIGNLEFT	沿窗口左边界排列条目
LVA_ALIGNTOP	沿窗口上边界排列条目
LVA_DEFAULT	根据列表视图的当前排列方式进行排列
LVA_SNAPTOGRID	将所有图标排列到距其最近的网格位置

• EditLable
调用该函数以开始在位编辑指定条目文本，其原型为：

```
CEdit* EditLabel( int nItem );
```

返回值：
如果函数调用成功，则返回用于编辑条目文本的 CEdit 对象，否则返回 NULL。

参数：
nItem ——指定了将被编辑的条目索引。

具有 LVS_EDITLABELS 风格的列表视图控件允许用户在位编辑条目文本。当用户单击具有输入焦点的某条目的文本时，即可开始在位编辑。

• DeleteColumn
调用该函数以删除列表视图控件中的指定列，其原型为：

```
BOOL DeleteColumn( int nCol );
```

返回值：
如果函数调用成功，则返回非零值，否则返回零值。

参数：
nCol ——指定了将被删除的列索引。

• CreateDragImage
调用该函数以为指定条目创建拖动图像列表，其原型为：

```
CImageList * CreateDragImage( int nItem, LPPOINT lpPoint );
```

返回值:

如果函数调用成功,则返回指向拖动图像列表的指针,否则返回 NULL。

参数:

nItem ——指定了将创建拖动图像列表的条目索引。

lpPoint ——指定了将接收图像初始位置左上角坐标的 POINT 结构地址。

CImageList 对象是持久对象,因此在使用完毕后必须将其删除,程序如下:

```
CImageList * pImageList = MyListCtrl.CreateDragImage(nItem,&point);  
...  
...  
delete pImageList;
```

• InsertColumn

调用该函数以向列表视图控件中插入新列,其原型为:

```
int InsertColumn( int nCol, const LVCOLUMN * pColumn );
```

返回值:

如果函数执行成功,则返回新列索引。

参数:

nCol ——指定了新列索引。

pColumn ——为 LVCOLUMN 结构指针,包含了将插入的列信息。LVCOLUMN 结构的定义如下:

```
typedef struct _LVCOLUMN {  
    UINT mask;  
    int fmt;  
    int cx;  
    LPTSTR pszText;  
    int cchTextMax;  
    int iSubItem;  
} LV_COLUMN;
```

结构成员:

mask ——指定了结构的有效成员,其取值如表 5-10 所示。

表 5-10 mask 成员取值

mask 成员取值	含义
LVCF_FMT	成员 fmt 有效
LVCF_SUBITEM	成员 iSubItem 有效
LVCF_TEXT	成员 pszText 有效
LVCF_WIDTH	成员 cx 有效

fmt ——指定了列的对齐方式,其取值如表 5-11 所示。

表 5-11 fmt 成员取值

fmt 成员取值	含义
LVCFMT_CENTER	居中排列
LVCFMT_LEFT	左对齐排列
LVCFMT_RIGHT	右对齐排列

- cx ——指定了列宽。
- pszText ——指定了列文本缓冲区地址。
- cchTextMax ——指定了 pszText 字符串的长度。
- iSubItem ——指定了条目子项索引。

5.1.4 虚函数

CListCtrl 类的虚函数为 DrawItem,调用该函数以绘制具有自绘制属性的列表视图控件。DrawItem 的原型为:

```
virtual void DrawItem( LPDRAWITEMSTRUCT lpDrawItemStruct );
```

参数:

lpDrawItemStruct ——为 DRAWITEMSTRUCT 结构。重载该函数可以按用户希望的方式绘制列表视图控件。

5.2 列表视图控件常用操作编程

列表视图控件的功能很强,使用时涉及的操作也很多,本节将介绍其常用操作编程方法。

5.2.1 创建列表视图控件

列表视图控件的创建方法取决于使用方式:是直接使用控件还是通过列表视图使用。如果使用列表视图,那么框架将以文档/视图的创建顺序首先创建视图,在视图被创建的同时列表视图控件也被创建(两者实际是同一对象)。控件的创建是在视图的 OnCreate 消息处理函数中完成的。在这种情况下,应用程序只需要通过 GetListCtrl 函数就可以得到控件指针,并通过该指针调用控件的成员函数对控件进行操作。

如果要在对话框中直接使用列表视图控件,一般需要遵循以下步骤:

- (1) 在对话框编辑器中,直接向对话框资源中添加列表视图控件,并指定控件 ID。
- (2) 使用 ClassWizard 为控件添加 CListCtrl 类型的成员变量。

(3) 使用 ClassWizard 在对话框类中添加需要处理的列表视图控件的消息映射。

(4) 在 OnInitDialog 函数中设置 CListCtrl 控件的属性。

如果是在非对话框窗口中使用列表视图控件,则一般要遵循以下步骤:

(1) 在视图或窗口类中定义列表视图控件变量。

(2) 在视图类的 OnInitialUpdate(或窗口类的 OnCreate)函数中调用控件的 Create 函数。

5.2.2 向控件中添加新条目和新列

在 CListCtrl 类中定义了好几个版本的 InsertItem 函数,以向控件中插入新条目。在执行插入操作时,根据当时拥有的信息选择合适的版本。通过设置 LV_ITEM 结构的成员变量,程序员可以很好地控制控件条目的属性。清单 5-1 为 AddItem 函数的源代码,其中封装了向列表视图控件中插入新条目的操作:

清单 5-1 AddItem() 函数

```
BOOL CMyListCtrl::AddItem(int nItem, int nSubItem, LPCTSTR strItem, int nImageIndex)
{
    LV_ITEM lvItem;
    lvItem.mask = LVIF_TEXT;
    lvItem.iItem = nItem;
    lvItem.iSubItem = nSubItem;
    lvItem.pszText = (LPTSTR) strItem;
    if(nImageIndex != -1){
        lvItem.mask |= LVIF_IMAGE;
        lvItem.iImage = LVIF_IMAGE;
    }
    if(nSubItem == 0)
        return InsertItem(&lvItem);
    return SetItem(&lvItem);
}
```

对于报表风格的列表视图来说,其中的列提供了将不同条目中的子条目组织起来的方法。这种组织是通过列和子条目间一一对应的关系实现的。例如 Windows 资源管理器,其中列表视图控件部分的第一列为文件夹、文件图标和标签;而其他列则给出文件尺寸、文件类型和最近修改时间等信息。

虽然随时都可以将列插入列表视图控件中,但是只有当控件的 LVS_REPORT 风格被设置时,新列才能被显示出来。每一列都有相关的标头(CHeaderCtrl)对象,该对象是列的标签并允许用户改变列的尺寸。

如果列表视图控件支持报表风格,则必须为每个可能的子条目都添加一列。添加列时首先需要设置 LV_COLUMN 结构,然后再调用 InsertColumn 函数。在添加了必需的列之后,可以使用嵌入标头对象对这些列进行记录。清单 5-2 所示为 AddColumn 函数的源代

码,其中封装了向列表视图控件中插入一系列的操作:

清单 5-2 AddColumn()函数

```
BOOL CMyListCtrl::AddColumn(LPCTSTR strItem,int nItem,int nSubItem,int nMask,
int nFmt)
{
    LV_COLUMN lvc;
    lvc.mask = nMask;
    lvc.fmt = nFmt;
    lvc.pszText = (LPTSTR) strItem;
    lvc.cx = GetStringWidth(lvc.pszText) + 65;
    if(nMask & LVCF_SUBITEM){
        if(nSubItem != -1)
            lvc.iSubItem = nSubItem;
        else
            lvc.iSubItem = nItem;
    }
    return InsertColumn(nItem,&lvc);
}
```

读者可以看到,实际上述这两个函数就是将原来向列表视图控件中添加行、列的一系列操作封装在一个函数中,忽略了麻烦的细节,而且使操作简单而直观。读者在开发程序时,也应该常常使用这种技巧,以提高效率。

5.2.3 改变控件的扩展风格

在创建了列表视图控件后,用户随时可以修改控件的窗口风格。在改变了窗口风格后,可以接着改变控件的视图类型。例如在资源管理器中提供了菜单命令和工具栏按钮来改变视图风格,如图标视图、列表视图等等。当用户选择了指定菜单命令后,应用程序可以调用 `GetWindowLong` 以得到控件的当前风格,然后调用 `SetWindowLong` 来设置新风格。

除了标准的列表视图控件风格外,MFC 还提供了扩展风格。使用扩展风格能够为控件提供很多很好的特性,参见 5.1.2 节中介绍的 `SetExtendedStyle` 函数。设置控件扩展风格的示范代码如下:

```
m_myListCtrl.SetExtendedStyle(LVS_EX_TRACKSELECT|LVS_EX_ONECLICKACTIVATE);
```

5.2.4 使用图像列表

列表视图控件中的每个条目是由图标、标签以及子条目组成的。其中条目图标包含于与控件相关的图像列表中。在第一个图像列表对象中,包括在图标视图下列表视图控件中将显示的全尺寸图标。在第二个(可选)图像列表对象中,包含在其他视图下控件中将显示的小尺寸图标。在第三个(可选)图像列表对象中,包含状态图像,例如复选框等。它们将在小图标的前面显示。在第四个(可选)图像列表对象中,包含在列表视图控件的

标头中显示的图像。

如果列表视图控件具有 LVS_SHAREIMAGELISTS 风格,则程序员必须在控件使用完毕后,手动销毁图像列表对象。一般在多个列表视图控件共用相同的图像列表时,使用上述风格。如果不需要在控件中显示图标,当然也可以不创建图像列表。

下面的示范代码,给出了将图像列表与列表视图控件相联系的方法:

```
// 创建并初始化图像列表,然后将其与列表视图控件相联系
m_pImageList = new CImageList();
ASSERT(m_pImageList != NULL);
m_pImageList->Create(32, 32, TRUE, 4, 4);
m_pImageList->Add(pApp->LoadIcon(IDI_ICONLIST1));
m_pImageList->Add(pApp->LoadIcon(IDI_ICONLIST2));
m_listctrl.SetImageList(m_pImageList, LVSIL_NORMAL);
```

5.2.5 操作控件的工作区域

默认情况下,列表视图控件将以标准网格风格排列其中的条目。然而还有另外一种方法,即工作区域用于将条目排列到矩形组中。只有当列表视图控件处于小图标或大图标模式时,工作区域才可见。当然控件处于报表或列表模式时,工作区域虽不可见但其数据依然被保存。

工作区域可以用于显示空边界(例如条目的上下左右),或使水平滚动条出现在通常不会出现的不地方。另一个用途就是在移动或拖动条目时,为其创建多个工作区域。这样,可以在单个视图中创建具有不同含义的区域。用户能够接着通过将条目放置到不同区域达到分类的目的。例如,在文件系统视图中可以有两个区域,其中一个显示可读写的文件,而另一个则显示只读文件。如果文件条目被移动到只读区域,则它将自动变成只读属性。而从只读区域将某文件条目移动到可读写区域,也将导致该文件的属性变为可读写。

列表视图控件为操作工作区域提供了三个成员函数。GetWorkAreas 和 SetWorkAreas 能够得到和设置 CRect 对象数组,而正是在这些数组中存储着当前工作区域。而 GetNumberOfWorkAreas 函数能够得到当前列表视图控件的工作区域数(默认为 0)。

当工作区域被创建时,处于工作区域中的条目就变成了它的成员。类似地,如果条目被移动到某个工作区域时,它也会变成该工作区域的成员。如果某个条目不处于任何工作区域,它将自动变成第一个工作区域的成员(索引号为 0)。如果希望将某个新建的条目放置到指定的工作区域,那么在创建该条目后应该调用 SetItemPosition 函数。下面的示范代码实现了 4 个工作区域(rcWorkAreas),它们尺寸相同并具有 10 像素宽的边界。

```
CRect curRect;
CSize size;
size = m_listctrl.ApproximateViewRect();
size.cx += 100;
size.cy += 100;

CRect rcWorkAreas[4];
```

```
rcWorkAreas[0].SetRect(0, 0, (size.cx / 2) - 5, (size.cy / 2) - 5);
rcWorkAreas[1].SetRect((size.cx / 2) + 5, 0, size.cx, (size.cy / 2) - 5);
rcWorkAreas[2].SetRect(0, (size.cy / 2) + 5, (size.cx / 2) - 5, size.cy);
rcWorkAreas[3].SetRect((size.cx / 2) + 5, (size.cy / 2) + 5, size.cx, size.cy);

//设置工作区域
m_listctrl.SetWorkAreas(4, rcWorkAreas);
```

调用 `ApproximateViewRect` 函数,以得到需要在第一个区域中显示所有条目需要的大概面积。接着以同样的方式估计其他 3 个区域,并将为它们添加 5 个像素宽的边界。

下面的示范代码将当前列表中的条目添加到不同的区域(`rcWorkAreas`)中,然后刷新控件显示。

```
// 为每个工作区域设置插入点
CPoint rgptWork[4];
for (int i = 0; i < 4; i++)
{
    rgptWork[i].x = rcWorkAreas[i].left + 10;
    rgptWork[i].y = rcWorkAreas[i].top + 10;
}

// 现在移动所有条目到不同的区域中
for (i = 0; i < 20; i++)
    m_listctrl.SetItemPosition(i, rgptWork[i % 4]);

// 强制控件重新排列其中的条目
m_listctrl.Arrange(LVA_DEFAULT);
```

5.2.6 虚列表控件

虚列表控件为具有 `LVS_OWNERDATA` 风格的列表视图控件。该风格允许控件的条目数可以达到 `DWORD` 数量级,而常规情况下条目数只能达到 `int` 数量级(指语言中的数据类型)。不过,该风格的重大优势在于它在任意时刻只将条目的一部分载入内存中。因此,应用程序可以使用虚列表控件来处理大量的数据库信息。除了在 `CListCtrl` 中提供了对虚列表控件的支持外,MFC 还在 `CListView` 中提供了相同的支持。

1. 处理 `LVN_GETDISPINFO` 通告消息

虚列表控件只需要维护很少的条目信息。除了条目选择和焦点信息,其他条目信息都由控件的父窗口负责维护。框架通过 `LVN_GETDISPINFO` 通告消息检索条目信息。虚列表控件的父窗口(或控件本身)必须处理该通告以提供信息。使用 `ClassWizard` 能够很容易地完成这个工作,结果代码如下所示:

```
BEGIN_MESSAGE_MAP(CMyListCtrl, CListCtrl)
    //{{AFX_MSG_MAP(CMyListCtrl)
    ON_NOTIFY_REFLECT(LVN_GETDISPINFO, OnGetdispinfo)
    //}}AFX_MSG_MAP
```



```
END_MESSAGE_MAP()
```

在 LVN_GETDISPINFO 消息的处理函数中,必须检查框架要检索的信息类型,可能的取值如表 5-12 所示:

表 5-12 框架检索的消息类型

消息类型常量	含义
LVIF_TEXT	返回 pszText 成员
LVIF_IMAGE	返回 iImage 成员
LVIF_INDENT	返回 iIndent 成员
LVIF_PARAM	返回 lParam 成员
LVIF_STATE	返回 state 成员

当确定了所检索的消息类型后,就应该将合适的信息返回给框架。下面的示范代码给出了提供条目文本和图像信息的例子:

```
LV_DISPINFO * pDispInfo = (LV_DISPINFO *)pNMHDR;
LV_ITEM * pItem = &(pDispInfo->item);

int iItemIdx = pItem->iItem;

if (pItem->mask & LVIF_TEXT) //valid text buffer?
{
    switch(pItem->iSubItem){
        case 0: //fill in main text
            lstrcpy(pItem->pszText,
                m_Items[iItemIdx].m_strItemText);
            break;
        case 1: //fill in sub item 1 text
            lstrcpy(pItem->pszText,
                m_Items[iItemIdx].m_strSubItem1Text);
            break;
        case 2: //fill in sub item 2 text
            lstrcpy(pItem->pszText,
                m_Items[iItemIdx].m_strSubItem2Text);
            break;
    }
}

if pItem->mask & LVIF_IMAGE) //valid image?
    pItem->iImage =
        m_Items[iItemIdx].m_iImageIndex;
```

2. 缓存和虚列表控件

由于虚列表控件经常用于处理大量数据,因此 MFC 推荐缓存被请求的条目数据,以提高检索性能。框架会提供缓存-提示机制,以辅助优化由 LVN_ODCACHEHINT 通告消息发送的缓存。程序员必须使用一些不同的方式来处理该通告消息,例如,使用 ClassWiz-

ard 重载控件对象的 OnChildNotify 函数,在函数中检查 LVN_ODCACHEHINT 消息,如果发现该消息则准备缓存,示范代码如下所示:

```
NMLVCACHEHINT * pcachehint = NULL;

if (message == WM_NOTIFY)
{
    NMHDR * phdr = (NMHDR *)lParam;

    switch(phdr->code)
    {
        case LVN_ODCACHEHINT:
            pcachehint = (NMLVCACHEHINT *) phdr;
            // 载入缓存
            PrepCache(pcachehint->iFrom, pcachehint->iTo);
            break;
        default:
            return CListCtrl::OnChildNotify(message, wParam, lParam, pLResult);
    }
    return FALSE;
}
else
    return CListCtrl::OnChildNotify(message, wParam, lParam, pLResult);
```

需要注意的是,如果消息类型不是 LVN_ODCACHEHINT,则使用控件的默认消息处理。

3. 查找指定条目

当需要查找某个指定条目时,虚列表控件会发送 LVN_ODFINDITEM 通告消息。当列表视图控件接收到快捷键存取或 LVM_FINDITEM 消息时,该通告就会被发送。检索信息将以 LVFINDINFO 结构的形式发送。在重载的 OnChildNotify 函数中检查 LVN_ODFINDITEM 消息,如果找到则进行条目查找(使用 CListCtrl 中的查找函数)。

5.3 一个经典话题

在不久前,重载 DrawItem 函数对列表视图控件进行自绘制操作以完成整行选择还是一个有趣而经典的编程话题,但是现在只需要为列表视图控件选择几个扩展风格选项(LVS_EX_FULLROWSELECT)就可以轻易地达到这个目的,一般使用 SetExtendedStyle 来完成扩展风格选项的设置,或是直接调用 ModifyStyleEx 函数修改控件的扩展风格。这些函数的使用方法参见 5.1 节。

虽然如此,本节还是要对这个主题进行讨论,希望读者能够仔细阅读,这对于理解 Windows 编程是很有帮助的。另外还要提醒读者:在程序的开发过程中,如果需要对控件或窗口实现某些特殊的效果,一定要首先仔细阅读联机帮助中有关该控件或窗口的创建

风格部分,以免进行无谓的劳动。

当列表视图控件为报表风格时,其中被选条目中只有第一列被高亮显示(如果不设置 LVS_EX_FULLROWSELECT 风格)。为了使整行被高亮显示,我们必须自己进行处理。实际上,Visual C++ 中早已给出了类似的示例(其名称为 ROWLIST)。但是,ROWLIST 示例在显示状态图像时会与文本叠加在一起;当将某列尺寸调整为 0 时(隐藏该行),该列中的内容将显示在下一行中。下面给出的代码是在研究了 ROWLIST 的基础上,修改了这些 bug 后,设计的改进列表视图管理类 CListViewEx。

(1) 创建具有 LVS_OWNERDRAWFIXED 风格的列表视图控件

在资源编辑器中创建列表视图控件时可以直接设置自绘制属性。如果使用的是 CListView 的派生类,那么可以在 PreCreateWindow 函数中设置 LVS_OWNERDRAWFIXED 风格,示范代码如清单 5-3 所示:

清单 5-3 PreCreateWindow() 函数

```
BOOL CListViewEx::PreCreateWindow(CREATESTRUCT& cs)
{
    // default is report view and full row selection
    cs.style &= LVS_TYPEMASK;
    cs.style &= LVS_SHOWSELALWAYS;
    cs.style |= LVS_REPORT | LVS_OWNERDRAWFIXED;
    cs.style |= LVS_EDITLABELS;
    return(CListView::PreCreateWindow(cs));
}
```

(2) 在类中添加枚举变量,用于标识是否高亮显示成员变量

在添加成员变量的时候,同时添加一个枚举变量,以标识不同类型的高亮选择状态。选择正常高亮状态(HIGHLIGHT_NORMAL)时,只有第一列的标签被高亮显示;而选择 HIGHLIGHT_ALLCOLUMNS 时,只有行中包括的所有列被高亮显示。也就是说,从最后一行到控件右边界的区域都不会被高亮显示;如果选择 HIGHLIGHT_ROW,则整行都会被高亮显示。可以在类的构造函数中初始化 m_nHighlight,以设置合适的选择状态。添加的变量定义如下所示:

```
public:
    enum EHighlight {HIGHLIGHT_NORMAL, HIGHLIGHT_ALLCOLUMNS, HIGHLIGHT_ROW};
protected:
    int m_nHighlight;
```

(3) 重载 DrawItem 函数

DrawItem 函数的第一部分代码在设置了一些变量后,首先保存当前设备环境,以便在函数完成时恢复原始设备环境,这是一个基本的编程规则。然后,检索条目的状态标志和图像,以便对其进行合适的绘制操作。

接下来的代码部分,首先检查条目是否需要被高亮显示。此时有两种可能:一种是条目应该取消其高亮选择,而另一种则是条目被选择应该高亮显示。默认情况下,只有当控件具有焦点或具有 LVS_SHOWSELALWAYS 风格时高亮显示。读者也许会注意到,在非自

绘制列表视图控件中,在控件失去焦点后,原有的高亮颜色会发生变化(通常为灰色)。在此处并没有实现这一特征,但是只要稍作改动就可以完成,读者可以自己试一试。接着,将文本偏移量设置为两倍的空格字符宽度,该偏移量将用于设置条目主项和子项两端的空白空间。

此后,函数计算高亮矩形尺寸,并设置设备环境以绘制高亮矩形、状态图像和标签(第一列的文本)。在进行任何实际的绘制之前,首先设置设备环境的裁剪区域,以使所有输出都显示在第一列中。对于条目图像来说,必须注意选择条目时使用的是 ILD_BLEND50 标志,这使得也同时选择高亮显示风格。如果不希望图像出现不同的显示,那么可以去除这一标志。

在绘制文本时,使用的是 DrawText 函数,它负责绘制所有列的文本。该函数的最后一个参数指定了绘制文本的方式。其中一个为 DT_END_ELLIPSIS,这会将指定字符串的一部分以椭圆形显示。这种方式可以使字符串与列宽相适合。在绘制完第一列的文本后,检查高亮标志。如果高亮标志标识只需将第一列高亮,则恢复原来的文本颜色和文本背景;否则使用同样的方式绘制其余列的文本。清单 5-4 所示为 DrawItem 函数的源代码:

清单 5-4 DrawItem() 函数

```
void CMyListCtrl::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
    CDC * pDC = CDC::FromHandle(lpDrawItemStruct->hDC);
    CRect rcItem(lpDrawItemStruct->rcItem);
    int nItem = lpDrawItemStruct->itemID;
    CImageList * pImageList;

    // 保存设备环境
    int nSavedDC = pDC->SaveDC();

    // 得到条目状态和图像
    LV_ITEM lvi;
    lvi.mask = LVIF_IMAGE | LVIF_STATE;
    lvi.iItem = nItem;
    lvi.iSubItem = 0;
    lvi.stateMask = 0xFFFF;
    GetItem(&lvi);

    // 条目是否应该被高亮显示
    BOOL bHighlight = ((lvi.state & LVIS_DROPHILITED) || ((lvi.state & LVIS_SELECTED)
        && ((GetFocus() == this) || (GetStyle() & LVS_SHOWSELALWAYS))));

    // 得到绘制矩形
    CRect rcBounds, rcLabel, rcIcon;
    GetItemRect(nItem, rcBounds, LVIR_BOUNDS);
    GetItemRect(nItem, rcLabel, LVIR_LABEL);
    GetItemRect(nItem, rcIcon, LVIR_ICON);
    CRect rcCol( rcBounds );

    CString sLabel = GetItemText( nItem, 0 );

    int offset = pDC->GetTextExtent(_T(" "), 1 ).cx * 2;

    CRect rcHighlight;
```

```

CRect rcWnd;
int nExt;
switch( m_nHighlight )
{
case 0:
    nExt = pDC->GetOutputTextExtent(sLabel).cx + offset;
    rcHighlight = rcLabel;
    if( rcLabel.left + nExt < rcLabel.right )
        rcHighlight.right = rcLabel.left + nExt;
    break;
case 1:
    rcHighlight = rcBounds;
    rcHighlight.left = rcLabel.left;
    break;
case 2:
    GetClientRect(&rcWnd);
    rcHighlight = rcBounds;
    rcHighlight.left = rcLabel.left;
    rcHighlight.right = rcWnd.right;
    break;
default:
    rcHighlight = rcLabel;
}

// 绘制背景颜色
if( bHighlight )
{
    pDC->SetTextColor(::GetSysColor(COLOR_HIGHLIGHTTEXT));
    pDC->SetBkColor(::GetSysColor(COLOR_HIGHLIGHT));

    pDC->FillRect(rcHighlight, &CBrush(::GetSysColor(COLOR_HIGHLIGHT)));
}
else
    pDC->FillRect(rcHighlight, &CBrush(::GetSysColor(COLOR_WINDOW)));

// 设置剪切区域
rcCol.right = rcCol.left + GetColumnWidth(0);
CRgn rgn;
rgn.CreateRectRgnIndirect(&rcCol);
pDC->SelectClipRgn(&rgn);
rgn.DeleteObject();

// 绘制状态图标
if (lvi.state & LVIS_STATEIMAGEMASK)
{
    int nImage = ((lvi.state & LVIS_STATEIMAGEMASK) >> 12) - 1;
    pImageList = GetImageList(LVSIL_STATE);
    if (pImageList)
    {
        pImageList->Draw(pDC, nImage, CPoint(rcCol.left, rcCol.top), ILD_
            TRANSPARENT);
    }
}

// 绘制正常和重叠图像

```

```

pImageList = GetImageList(LVSIL_SMALL);
if (pImageList)
{
    UINT nOvlImageMask = lvi.state & LVIS_OVERLAYMASK;
    pImageList->Draw(pDC, lvi.iImage, CPoint(rcIcon.left, rcIcon.top),
        (bHighlight? ILD_BLEND50:0) | ILD_TRANSPARENT | nOvlImageMask);
}

// 绘制第 0 列
rcLabel.left += offset/2;
rcLabel.right -= offset;

pDC->DrawText (sLabel, -1, rcLabel, DT_LEFT | DT_SINGLELINE | DT_NOPREFIX | DT_NOCLIP | DT_VCENTER | DT_END_ELLIPSIS);

// 绘制其他列
LV_COLUMN lvc;
lvc.mask = LVCF_FMT | LVCF_WIDTH;

if( m_nHighlight == 0 ) // 高亮显示第一列
{
    pDC->SetTextColor(::GetSysColor(COLOR_WINDOWTEXT));
    pDC->SetBkColor(::GetSysColor(COLOR_WINDOW));
}

rcBounds.right = rcHighlight.right > rcBounds.right ? rcHighlight.right :
    rcBounds.right;
rgn.CreateRectRgnIndirect(&rcBounds);
pDC->SelectClipRgn(&rgn);

for(int nColumn = 1; GetColumn(nColumn, &lvc); nColumn++)
{
    rcCol.left = rcCol.right;
    rcCol.right += lvc.cx;

    // 如果需要绘制背景
    if( m_nHighlight == HIGHLIGHT_NORMAL )
        pDC->FillRect(rcCol, &CBrush(::GetSysColor(COLOR_WINDOW)));

    sLabel = GetItemText(nItem, nColumn);
    if (sLabel.GetLength() == 0)
        continue;

    UINT nJustify = DT_LEFT;
    switch(lvc.fmt & LVCFMT_JUSTIFYMASK)
    {
    case LVCFMT_RIGHT:
        nJustify = DT_RIGHT;
        break;
    case LVCFMT_CENTER:
        nJustify = DT_CENTER;
        break;
    default:
        break;
    }

    rcLabel = rcCol;

```



```

        rcLabel.left += offset;
        rcLabel.right -= offset;

        pDC->DrawText(sLabel, -1, rcLabel, nJustify | DT_SINGLELINE |
            DT_NOPREFIX | DT_VCENTER | DT_END_ELLIPSIS);
    }

    // 如果条目具有焦点,则绘制焦点矩形
    if (lvi.state & LVIS_FOCUSED && (GetFocus() == this))
        pDC->DrawFocusRect(rcHighlight);

    // 恢复设备环境
    pDC->RestoreDC( nSavedDC );
}

```

(4) 完成 RepaintSelectedItems 辅助函数

设计这个辅助函数的目的是添加或去除条目的焦点矩形。该函数同时还能根据控件是否具有 LVS_SHOWSELALWAYS 风格决定添加或去除被选条目的高亮状态。该函数实际上完成的工作只是设置一些标志,而绘制工作则是由 DrawItem 完成的。清单 5-5 所示为 RepaintSelectedItems 函数的源代码:

清单 5-5 RepaintSelectedItems() 函数

```

void CMyListCtrl::RepaintSelectedItems()
{
    CRect rcBounds, rcLabel;

    int nItem = GetNextItem( -1, LVNI_FOCUSED);

    if(nItem != -1)
    {
        GetItemRect(nItem, rcBounds, LVIR_BOUNDS);
        GetItemRect(nItem, rcLabel, LVIR_LABEL);
        rcBounds.left = rcLabel.left;

        InvalidateRect(rcBounds, FALSE);
    }

    if(! (GetStyle() & LVS_SHOWSELALWAYS))
    {
        for(nItem = GetNextItem( -1, LVNI_SELECTED);
            nItem != -1; nItem = GetNextItem(nItem, LVNI_SELECTED))
        {
            GetItemRect(nItem, rcBounds, LVIR_BOUNDS);
            GetItemRect(nItem, rcLabel, LVIR_LABEL);
            rcBounds.left = rcLabel.left;

            InvalidateRect(rcBounds, FALSE);
        }
    }

    UpdateWindow();
}

```

(5) 在 OnPaint 函数中添加代码以保证重绘整行

当列表控件重新绘制某条目时,它只重新绘制条目所占的区域。也就是说,如果最后一列的右边界小于控件的右边界的话,这部分空间不会被重新绘制。这时,如果需要高亮显示整行,则需要同时重新绘制这块空白区域。清单 5-6 所示为 OnPaint 函数的源代码:

清单 5-6 OnPaint() 函数

```
void CMyListCtrl::OnPaint()
{
    // 如果是整行选择模式,则应该扩展裁剪区域
    if (m_nHighlight == HIGHLIGHT_ROW &&
        (GetStyle() & LVS_TYPEMASK) == LVS_REPORT)
    {
        CRect rcBounds;
        GetItemRect(0, rcBounds, LVIR_BOUNDS);

        CRect rcClient;
        GetClientRect(&rcClient);
        if(rcBounds.right < rcClient.right)
        {
            CPaintDC dc(this);

            CRect rcClip;
            dc.GetClipBox(rcClip);

            rcClip.left = min(rcBounds.right - 1, rcClip.left);
            rcClip.right = rcClient.right;

            InvalidateRect(rcClip, FALSE);
        }
    }

    CListCtrl::OnPaint();
}
```

(6) 处理 WM_KILLFOCUS 和 WM_SETFOCUS 消息

当控件失去焦点时,被选条目的焦点矩形应该去除。而当控件重新得到焦点时,应该重新绘制焦点矩形。这就需要添加 WM_KILLFOCUS 和 WM_SETFOCUS 消息的处理函数,其中都会调用 RepaintSelectedItems 辅助函数。清单 5-7 和 5-8 所示为 WM_KILLFOCUS 和 WM_SETFOCUS 的消息处理函数:

清单 5-7 OnKillFocus() 函数

```
void CMyListCtrl::OnKillFocus(CWnd* pNewWnd)
{
    CListCtrl::OnKillFocus(pNewWnd);

    // 检查是否焦点由标签编辑框得到
    if(pNewWnd != NULL && pNewWnd->GetParent() == this)
        return;

    // 否则应该重新绘制条目,以改变显示
```

```

        if((GetStyle() & LVS_TYPEMASK) == LVS_REPORT)
            RepaintSelectedItems();
    }

```

清单 5-8 OnSetFocus() 函数

```

void CMyListCtrl::OnSetFocus(CWnd* pOldWnd)
{
    CListCtrl::OnSetFocus(pOldWnd);

    // 检查是否由条目标签编辑框得到焦点
    if(pOldWnd != NULL && pOldWnd->GetParent() == this)
        return;

    // 否则应该重新绘制条目,以改变显示
    if((GetStyle() & LVS_TYPEMASK) == LVS_REPORT)
        RepaintSelectedItems();
}

```

(7) 添加修改选择模式的辅助函数 SetHighlightType

在此函数中应该更新成员变量 `m_nHighlight` 并重绘控件,以便条目以合适的高亮模式显示。清单 5-9 所示为 `SetHighlightType` 函数的源代码:

清单 5-9 SetHighlightType() 函数

```

int CMyListCtrl::SetHighlightType(EHighlight hilite)
{
    int oldhilite = m_nHighlight;
    if( hilite <= HIGHLIGHT_ROW )
    {
        m_nHighlight = hilite;
        Invalidate();
    }
    return oldhilite;
}

```

`AddItem` 函数和 `AddColumn` 函数实现了增强的向列表视图控件中插入行和列的功能,清单 5-10 和清单 5-11 列出了 `AddItem` 和 `AddColumn` 函数的源代码。

清单 5-10 AddItem() 函数

```

BOOL CMyListCtrl::AddItem(int nItem, int nSubItem, LPCTSTR strItem, int nImageIndex)
{
    LV_ITEM lvItem;
    lvItem.mask = LVIF_TEXT;
    lvItem.iItem = nItem;
    lvItem.iSubItem = nSubItem;
    lvItem.pszText = (LPTSTR) strItem;
    if(nImageIndex != -1){
        lvItem.mask |= LVIF_IMAGE;
        lvItem.iImage = LVIF_IMAGE;
    }
}

```

```

    if(nSubItem == 0)
        return InsertItem(&lvItem);
    return SetItem(&lvItem);
}

```

清单 5-11 AddColumn() 函数

```

BOOL CMyListCtrl::AddColumn(LPCTSTR strItem, int nItem, int nSubItem, int nMask,
                           int nFmt)
{
    LV_COLUMN lvc;
    lvc.mask = nMask;
    lvc.fmt = nFmt;
    lvc.pszText = (LPTSTR) strItem;
    lvc.cx = GetStringWidth(lvc.pszText) + 65;
    if(nMask & LVCF_SUBITEM){
        if(nSubItem != -1)
            lvc.iSubItem = nSubItem;
        else
            lvc.iSubItem = nItem;
    }
    return InsertColumn(nItem, &lvc);
}

```

读者可以看到,实际上这两个函数就是将原来向列表视图控件中添加行、列的一系列操作封装在一个函数中,忽略了麻烦的细节,而且使操作简单而直观。读者在开发程序时,也应该常常使用这种技巧,以提高效率。

5.4 动态改变列表视图的行高

当应用程序改变了列表视图控件所用的字体时,控件或其父窗口并不能自动改变行的高度以适应新的字体。如果新字体较小,那么问题还不大,而如果新字体比原来字体大的话,就会使列表视图控件中的内容被裁剪而显示不全。那么应该如何解决这个问题呢?我们知道当控件被创建时,其父窗口都会向其发送 WM_MEASUREITEM 消息,而当控件接受到该消息时,将使用其尺寸信息填充 MEASUREITEMSTRUCT 结构,并将其返回父窗口。这样,我们只要在应用程序改变列表视图控件所用的字体时,强制其发送 WM_MEASUREITEM 消息即可。当然,我们希望现在做的工作以后能够很方便地使用,那么最好的方式就是创建 CListCtrl 的派生类(下面使用的 CMyListCtrl 类),并在该类中封装以上处理。要达到这一目的,需要以下步骤:

(1) 添加 WM_SETFONT 消息处理函数。前面已经说过当控件被创建时,父窗口会向其发送 WM_MEASUREITEM 消息。不过当控件的尺寸改变时,也会触发该消息。既然我们希望当字体改变时,列表视图控件的行高也改变,那么这个触发当然最好在 WM_SETFONT 消息处理函数 OnSetFont 中进行。此时只要“欺骗”Windows,使它认为控件的尺寸被

改变即可。

由于 ClassWizard 中并没有列出 WM_SETFONT 消息,因此必须手动在消息映射中添加。不过手动添加的代码必须在 AFX_MSG 括号外,否则以后使用 ClassWizard 对应用程序的任何改变都将导致手动修改丢失。将 WM_SETFONT 添加到消息映射中的形式如下所示:

```
// .h 头文件中
//{{AFX_MSG(CMyListCtrl)
:
:
//}}AFX_MSG
afx_msg LRESULT OnSetFont(WPARAM wParam, LPARAM);
DECLARE_MESSAGE_MAP()

////////////////////////////////////
// .h 源文件中
BEGIN_MESSAGE_MAP(CMyListCtrl, CListCtrl)
//{{AFX_MSG_MAP(CMyListCtrl)
:
:
//}}AFX_MSG_MAP
ON_MESSAGE(WM_SETFONT, OnSetFont)
END_MESSAGE_MAP()
```

在 OnSetFont 函数中通过调用 SendMessage 函数,向控件发送 WM_WINDOWPOSCHANGED 消息,并触发 WM_MEASUREITEM 消息。发送 WM_WINDOWPOSCHANGED 消息时,并没有指定 SWP_NOSIZE 标志(该标志表示窗口尺寸没有改变,从而不会触发 WM_MEASUREITEM 消息),因此需要将 WINDOWPOS 结构中的宽度和高度成员设置为实际尺寸。清单 5-12 所示为 OnSetFont 函数的源代码:

清单 5-12 OnSetFont()函数

```
LRESULT CMyListCtrl::OnSetFont(WPARAM wParam, LPARAM)
{
    LRESULT res = Default();

    CRect rc;
    GetWindowRect( &rc );

    WINDOWPOS wp;
    wp.hwnd = m_hWnd;
    wp.cx = rc.Width();
    wp.cy = rc.Height();
    wp.flags = SWP_NOACTIVATE | SWP_NOMOVE | SWP_NOOWNERZORDER |
               SWP_NOZORDER;
    SendMessage( WM_WINDOWPOSCHANGED, 0, (LPARAM)&wp );
    return res;
}
```

(2) 添加 WM_MEASUREITEM 消息处理函数。由于 ClassWizard 同样也没有为该消息提供映射,因此也需手动添加,添加方式与添加 WM_FONT 相同:


```

// .h 头文件中
//{{AFX_MSG(CMyListCtrl)
:
:
//}}AFX_MSG
afx_msg LRESULT OnSetFont(WPARAM wParam, LPARAM);
afx_msg void MeasureItem ( LPMEASUREITEMSTRUCT lpMeasureItemStruct );
DECLARE_MESSAGE_MAP()

////////////////////////////////////
// .h 源文件中
BEGIN_MESSAGE_MAP(CMyListCtrl, CListCtrl)
//{{AFX_MSG_MAP(CMyListCtrl)
:
:
//}}AFX_MSG_MAP
ON_MESSAGE(WM_SETFONT, OnSetFont)
ON_WM_MEASUREITEM_REFLECT( )
END_MESSAGE_MAP()

```

在 WM_MEASUREITEM 的消息响应函数 MeasureItem 中,根据当前的字体高度设置列表视图的行高。清单 5-13 所示为 WM_MEASUREITEM 的消息响应函数:

清单 5-13 MeasureItem() 函数

```

void CMyListCtrl::MeasureItem ( LPMEASUREITEMSTRUCT lpMeasureItemStruct )
{
    LOGFONT lf;
    GetFont() -> GetLogFont( &lf );

    if( lf.lfHeight < 0 )
        lpMeasureItemStruct->itemHeight = -lf.lfHeight;
    else
        lpMeasureItemStruct->itemHeight = lf.lfHeight;
}

```

读者可以看到,行高的设置实际是通过改变 MEASUREITEMSTRUCT 结构的 itemHeight 成员完成的,该结构的定义如下:

```

typedef struct tagMEASUREITEMSTRUCT {
    UINT CtlType;
    UINT CtlID;
    UINT itemID;
    UINT itemWidth;
    UINT itemHeight;
    DWORD itemData
} MEASUREITEMSTRUCT;

```

结构成员:

CtlType ——指定了控件类型,其取值如表 5-13 所示。

表 5-13 CtlType 成员取值

CtlType 成员取值	含义
ODT_COMBOBOX	自绘制组合框
ODT_LISTBOX	自绘制列表框
ODT_BUTTON	自绘制按钮控件
ODT_LISTVIEW	自绘制列表视图控件
ODT_MENU	自绘制菜单项

CtrlID ——指定了需要自绘制的控件 ID,而对于菜单项则无需使用该成员。

itemID ——可以为菜单项 ID、列表框或组合框中某项的索引。对于空列表框或组合框,该成员为负值,这时应用程序只绘制焦点矩形(其坐标由 rcItem 成员给出)。虽然此时控件中没有需要显示的项,但绘制焦点矩形还是很有必要的,因为这能够提示用户该控件是否具有输入焦点。当然,也可以设置 itemAction 成员为合适的值,使得无需绘制输入焦点。

itemWidth ——指定了菜单项的宽度,自绘制菜单项的父窗口必须在通告返回时,设置该成员。

itemHeight ——指定了菜单项的高度,自绘制菜单项的父窗口必须在通告返回时,设置该成员。

itemData ——对于列表框或组合框,该成员的取值可以为由 CComboBox::AddString、CComboBox::InsertString、CListBox::AddString 或 CListBox::InsertString 等函数传递给控件的值。

对于菜单项,该成员取值可以为由 CMenu::AppendMenu、CMenu::InsertMenu 或 CMenu::ModifyMenu 等函数传递给菜单的值。

5.5 改变列表视图控件的背景

默认情况下,列表视图控件的背景为白色,在本节中将向读者介绍如何改变控件的背景,从而使控件更具有吸引力。

5.5.1 改变背景颜色

如果设置了自绘制属性,那么改变列表视图控件的背景颜色是非常简单的。实际上只要在绘制行的某个元素之前,在重载的 DrawItem 函数中使用所需的颜色填充行的边界矩形即可。需要改变的代码如下所示:

```
// 绘制背景颜色
if( bHighlight )
{
    pDC-> SetTextColor(::GetSysColor(COLOR_HIGHLIGHTTEXT));
    pDC-> SetBkColor(::GetSysColor(COLOR_HIGHLIGHT));

    pDC-> FillRect(rcHighlight, &CBrush(::GetSysColor(COLOR_HIGHLIGHT)));
}
```

```

else
    pDC->FillRect(rcHighlight, &CBrush(::GetSysColor(COLOR_WINDOW)));

```

下面的代码则用于设置(绘制)控件背景:

```

// 绘制背景颜色
if( bHighlight )
{
    pDC->SetTextColor(::GetSysColor(COLOR_HIGHLIGHTTEXT));
    pDC->SetBkColor(::GetSysColor(COLOR_HIGHLIGHT));
    pDC->FillRect(rcHighlight, &CBrush(::GetSysColor(COLOR_HIGHLIGHT)));
}
else
{
    CRect rcClient, rcRow = rcItem;
    GetClientRect(&rcClient);
    rcRow.right = rcClient.right;

    pDC->FillRect(rcRow, &CBrush(RGB(255,255,0)));
}

```

此外,还要响应 WM_ERASEBKGD 消息。每当窗口的背景需要重新绘制时,都会发送该消息。这个处理是必要的,因为 DrawItem 函数只是针对某行的。清单 5-14 所示为 WM_ERASEBKGD 的消息响应函数的源代码:

清单 5-14 OnEraseBkgnd() 函数

```

BOOL CMyListCtrl::OnEraseBkgnd(CDC * pDC)
{
    CRect rcClient;
    GetClientRect( &rcClient );

    pDC->FillRect(rcClient, &CBrush(RGB(255,255,0)));
    return TRUE;
}

```

图 5-2 所示即为具有不同背景颜色的列表视图控件:

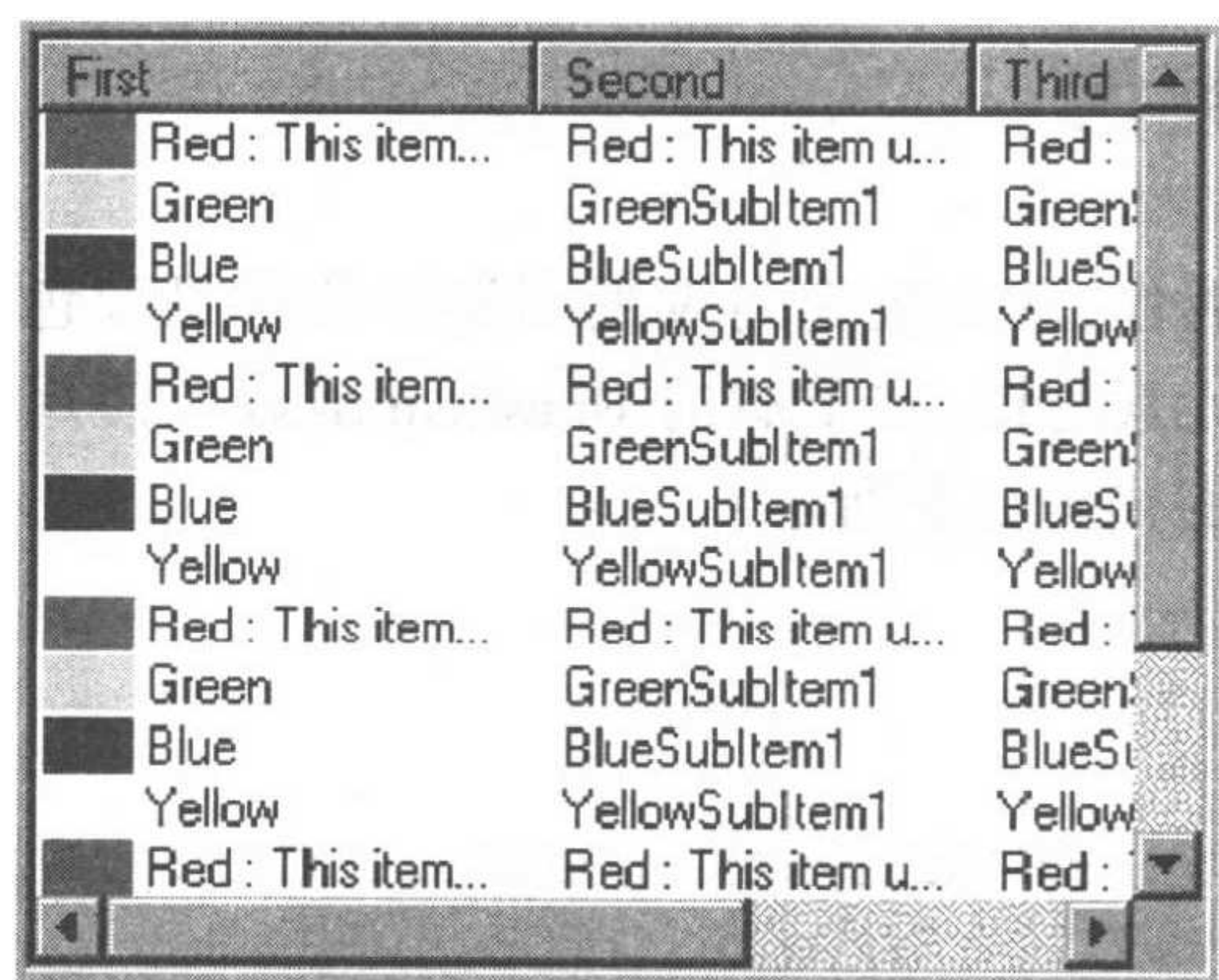


图 5-2 具有不同背景颜色的列表视图控件

5.5.2 使用位图背景

显然,使用位图作为列表视图控件的背景能够大大改善控件的外观,如图 5-3 所示。在大部分情况下,这种方式不但要比改变控件的背景颜色效果要好,而且其实现也相对要复杂一些,因此读者也能够从中学到更多的技巧。设置位图为控件背景时,控件一般应该具有自绘制类型。然而,控件如果不具有自绘制类型也可以,不过这将会影响控件的显示,因为无法根据控件状态对其进行定制修改。这就像简单地将位图和控件组合起来,而不是将其作为一个整体。

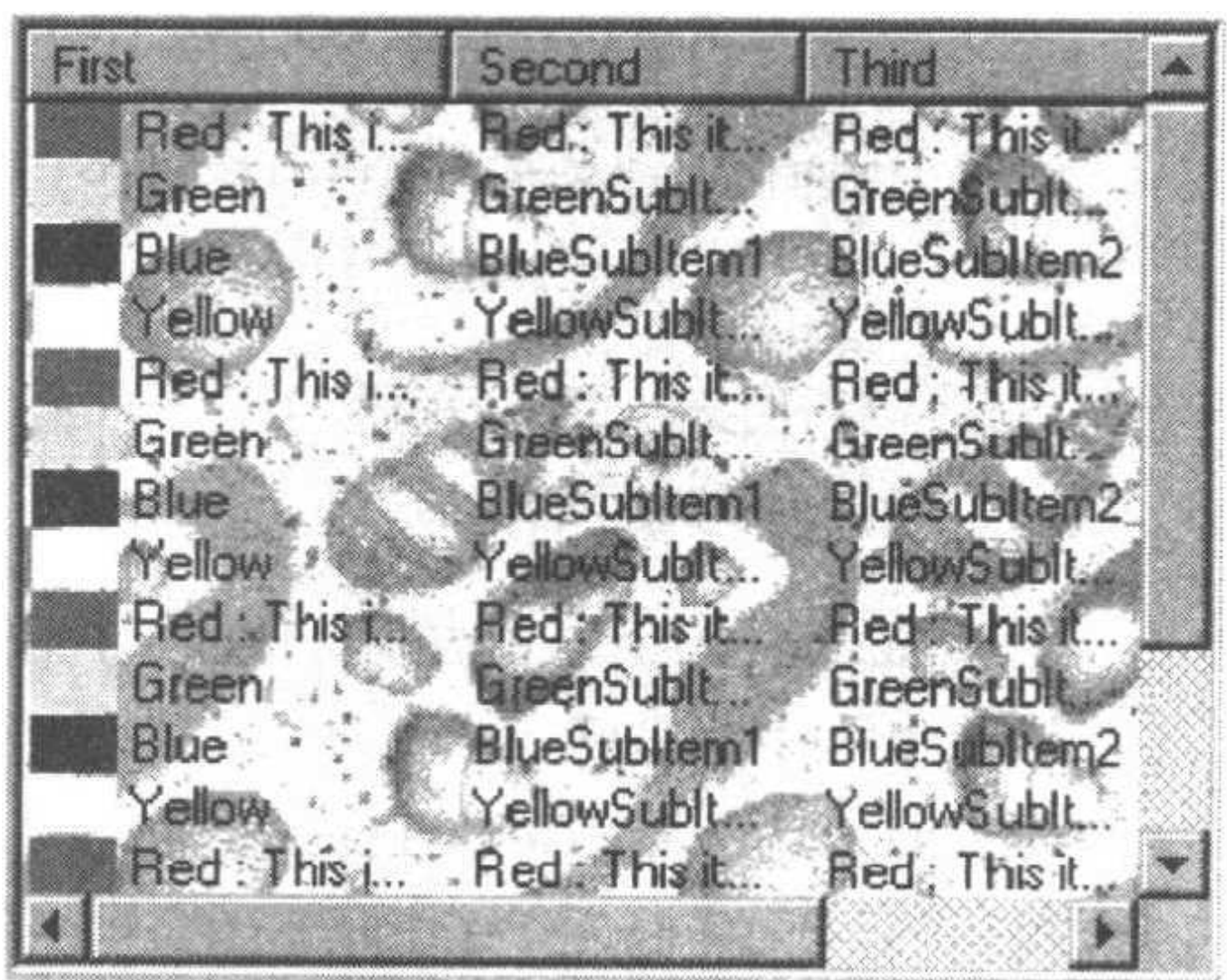


图 5-3 具有位图背景的列表视图控件

下面将向读者介绍如何使用 256 色位图(被添加为应用程序的位图资源)作为列表视图控件的背景。在选择位图时,需要注意不要使位图影响用户对文本的阅读。如果位图比控件的尺寸小,则它将在控件的客户区平铺显示。为了快速重绘,位图将与条目一起滚动。

为列表视图控件添加位图背景的步骤如下:

(1) 将列表视图控件设置为自绘制类型

这既可以在资源编辑器中完成,也可以在调用 Create 函数创建控件时完成,此外还可以实时调用 Modify 函数完成。

(2) 在控件管理类定义中添加成员变量

显然,每当某个条目需要重绘时就重新载入位图或创建逻辑调色板的做法效率非常低下,因此,在类定义中为其添加成员变量以存储位图、位图尺寸和调色板:

```
protected:
    CPalette m_pal;
    CBitmap m_bitmap;
    int m_cxBitmap, m_cyBitmap;
```

(3) 添加设置背景位图的成员函数

由于我们使用的位图是应用程序的资源,那么就有两种可能的调用方式:使用其 ID 或名称。因此在控件管理类中相应添加了两个不同版本的函数,其中之一将资源 ID 作为参数,而另一个则将资源名称作为参数。这两个函数的源代码如清单 5-15 所示:

清单 5-15 SetBkImage() 函数

```
BOOL CMyListCtrl::SetBkImage(UINT nIDResource)
{
    return SetBkImage( (LPCTSTR)nIDResource );
}

BOOL CMyListCtrl::SetBkImage(LPCTSTR lpszResourceName)
```

```

// 如果以前存在位图背景,则删除之
if( m_bitmap.m_hObject != NULL )
    m_bitmap.DeleteObject();
if( m_pal.m_hObject != NULL )
    m_pal.DeleteObject();

HBITMAP hBmp = (HBITMAP)::LoadImage( AfxGetInstanceHandle(),
    lpszResourceName, IMAGE_BITMAP, 0,0, LR_CREATEDIBSECTION );

if( hBmp == NULL )
    return FALSE;

m_bitmap.Attach( hBmp );
BITMAP bm;
m_bitmap.GetBitmap( &bm );
m_cxBitmap = bm.bmWidth;
m_cyBitmap = bm.bmHeight;

// 创建位图的逻辑调色板
DIBSECTION ds;
BITMAPINFOHEADER &bmInfo = ds.dsBmih;
m_bitmap.GetObject( sizeof(ds), &ds );

int nColors = bmInfo.biClrUsed ? bmInfo.biClrUsed : 1 << bmInfo.biBit-
    Count;

// 如果颜色数小于 256 则创建中间色调色板
CClientDC dc(NULL);
if( nColors > 256 )
    m_pal.CreateHalftonePalette( &dc );
else
{
    // 创建调色板
    RGBQUAD *pRGB = new RGBQUAD[nColors];
    CDC memDC;
    memDC.CreateCompatibleDC(&dc);

    memDC.SelectObject( &m_bitmap );
    ::GetDIBColorTable( memDC, 0, nColors, pRGB );

    UINT nSize = sizeof(LOGPALETTE) + (sizeof(PALETTEENTRY) * nColors);
    LOGPALETTE *pLP = (LOGPALETTE *) new BYTE[nSize];

    pLP->palVersion = 0x300;
    pLP->palNumEntries = nColors;

    for( int i = 0; i < nColors; i++ )
    {
        pLP->palPalEntry[i].peRed = pRGB[i].rgbRed;
        pLP->palPalEntry[i].peGreen = pRGB[i].rgbGreen;
        pLP->palPalEntry[i].peBlue = pRGB[i].rgbBlue;
    }
}

```



```

        pLP->palPalEntry[i].peFlags = 0;
    }

    m_pal.CreatePalette( pLP );

    delete[] pLP;
    delete[] pRGB;
}

Invalidate();

return TRUE;
}

```

在应用程序运行时,可以调用这两个函数随时改变位图背景。函数将首先删除原有背景然后设置新背景。接着,从应用程序资源中载入位图并将其与 CBitmap 对象相联系。载入位图资源时,我们选择全局函数::LoadImage 而不是 CBitmap::LoadBitmap。这是为了获得位图的 DIBSECTION,以便创建位图颜色的逻辑调色板(如果不使用逻辑调色板,那么在 256 色的显示器上位图可能会非常黯淡,当然如果显示器支持 64K 以上颜色就不会有这个问题)。同时函数将位图的尺寸保存起来以备后用。

在设置了位图后,就可以使用它来创建逻辑调色板了。首先调用 CBitmap::GetObject 函数得到 DIBSECTION 确定位图使用的颜色。有时 DIBSECTION 的 BITMAPINFOHEADER 部分并没有指定所使用的颜色。在这种情况下,就必须从位图每个像素所使用的位数来推断其所使用的颜色数。例如,8 位能够表示 256 种颜色,而 16 位能表示 64000 种颜色。

如果位图使用的颜色超过 256 种,那么它就没有颜色表。这时,只需要简单地创建与设备环境兼容的中间色调色板即可。中间色调色板实际就是包含所有不同颜色的调色板,它实际不是最好的方法但却是最方便的。

如果位图使用 256 色或更少的颜色,那么就无需创建调色板。这时,首先分配足以容纳颜色表的空间,然后调用::GetDIBColorTable 从位图中得到其颜色表。此外,还需要分配足够的空间用于创建逻辑调色板,并将位图颜色表中的颜色条目拷贝到调色板中。其中 palVersion 应该为 0x300。

在创建了 Cpalette 对象后,将以前分配的内存释放,并重绘窗口以便显示更新后的图像。

(4) 修改 DrawItem 函数以绘制位图

DrawItem 函数负责绘制列表中的每一个条目和背景位图。由于图像应该完全覆盖整个客户区域,因此应该调整裁剪区域以使其延展到客户区的右边界。同样,如果正在绘制列表控件中的最后一个条目,则裁剪区域应该扩展到客户区的下边界。而接下来就应该选择逻辑调色板,当然只有当设备支持调色板时该操作才有意义。

接下来,函数以平铺方式绘制位图,这时应该将第一个条目的左上角作为参照。这使图像会随着列表视图控件内容一起滚动。此外未选条目的背景只有在位图未被绘制时才被显示。DrawItem 函数的源代码如清单 5-16 所示,其中只列出了需要添加的部分:

清单 5-16 DrawItem() 函数

```

void CMyListCtrl::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{

```

```

...

// 如果需要绘制位图背景
if( m_bitmap.m_hObject != NULL )
{
    CDC tempDC;
    tempDC.CreateCompatibleDC(pDC);
    tempDC.SelectObject( &m_bitmap );

    GetClientRect(&rcClient);

    CRgn rgnBitmap;
    CRect rcTmpBmp( rcItem );

    rcTmpBmp.right = rcClient.right;

    // 检查最后一个条目的位置
    if( nIndex == GetItemCount() - 1 )
        rcTmpBmp.bottom = rcClient.bottom;

    rgnBitmap.CreateRectRgnIndirect(&rcTmpBmp);
    pDC->SelectClipRgn(&rgnBitmap);
    rgnBitmap.DeleteObject();

    if( pDC->GetDeviceCaps(RASTERCAPS) & RC_PALETTE && m_pal.m_hObject !=
        NULL )
    {
        pDC->SelectPalette( &m_pal, FALSE );
        pDC->RealizePalette();
    }

    CRect rcFirstItem;
    GetItemRect(0, rcFirstItem, LVIR_BOUNDS);
    for( int i = rcFirstItem.left; i < rcClient.right; i += m_cxBitmap )
        for( int j = rcFirstItem.top; j < rcClient.bottom; j += m_cyBitmap )
            pDC->BitBlt( i, j, m_cxBitmap, m_cyBitmap, &tempDC,
                        0, 0, SRCCOPY );
}

...
}

```

(5) 添加 WM_ERASEBKGND 消息的处理函数

当使用位图作为背景时,擦除它是毫无意义的,这只会产生闪烁的显示。因此,在使用位图作为背景时该消息的处理函数应该返回 TRUE。清单 5-17 所示为 WM_ERASEBKGND 消息的处理函数 OnEraseBkgnd 的源代码:

清单 5-17 OnEraseBkgnd()函数

```

BOOL CMyListCtrl::OnEraseBkgnd(CDC * pDC)
{
    if( m_bitmap.m_hObject != NULL )
        return TRUE;
    return CListCtrl::OnEraseBkgnd(pDC);
}

```

(6) 重载 OnNotify 函数以处理列的尺寸改变

当列表视图中的某列尺寸被修改时,只有新近露出的区域被重新绘制,这使背景位图产生难看的变化。因此,在某列的尺寸发生变化时,应该总是使列表视图控件的最右端被绘制,在实现这点,要实现这点,只要使用 HDN_ITEMCHANGING 通告消息即可。清单5-18所示为 OnNotify 函数的源代码:

清单 5-18 OnNotify() 函数

```

BOOL CMyListCtrl::OnNotify(WPARAM wParam, LPARAM lParam, LRESULT* pResult)
{
    HD_NOTIFY *pHDN = (HD_NOTIFY*)lParam;

    if(pHDN->hdr.code == HDN_ITEMCHANGINGW || pHDN->hdr.code == HDN_ITEMCHANGINGA)
    {
        if(m_bitmap.m_hObject != NULL)
        {
            CRect rcClient;
            GetClientRect(&rcClient);
            DWORD dwPos = GetMessagePos();
            CPoint pt( LOWORD(dwPos), HIWORD(dwPos) );
            ScreenToClient(&pt);
            rcClient.left = pt.x;
            InvalidateRect(&rcClient);
        }
    }
    return CListCtrl::OnNotify(wParam, lParam, pResult);
}

```

在上面的代码中首先调用 GetClientRect 得到整个控件客户区矩形,然后调用 GetMessagePos 函数得到导致通告产生的位置(列的左上角),并将该点的横坐标赋予客户区矩形的左上角横坐标。这样,将设置后的矩形传递给 InvalidateRect 函数时,被绘制的既不只是单列,也不是整个客户区,而是从尺寸被修改的列到控件右边界的区域。

(7) 添加 WM_QUERYNEWPALETTE 和 WM_PALETTECHANGED 消息处理函数

当控件将得到输入焦点时,会向窗口发送 WM_QUERYNEWPALETTE 消息,这使得窗口重新确认逻辑调色板,这样它能够使用其最佳的形式。当系统调色板改变时,控件会向窗口发送 WM_PALETTECHANGED 消息。如果不处理这些消息,那么当另一个应用程序改变了系统调色板后,本应用程序的背景图像可能会非常难看。而且非常不幸的是,这两个消息都是被发送给最顶层窗口的。

为了处理这个问题,OnQueryNewPalette 函数首先检查是否需要重新选择调色板。确定了逻辑调色板后,如果其中任何颜色被重新映射,则将重新绘制窗口。如果是列表视图控件自己改变了调色板,OnPaletteChanged 函数将不作任何事情,而是接着调用 OnQueryNewPalette 函数重新设置调色板。清单 5-19 和 5-20 所示为 OnQueryNewPalette 和 OnPaletteChanged 函数的源代码:

清单 5-19 OnQueryNewPalette() 函数

```

BOOL CMyListCtrl::OnQueryNewPalette()
{
    CClientDC dc(this);
    if( dc.GetDeviceCaps(RASTERCAPS) & RC_PALETTE && m_pal.m_hObject != NULL )
    {
        dc.SelectPalette( &m_pal, FALSE );
        BOOL result = dc.RealizePalette();
        if( result )
            Invalidate();
        return result;
    }

    return CListCtrl::OnQueryNewPalette();
}

```

清单 5-20 OnPaletteChanged() 函数

```

void CMyListCtrl::OnPaletteChanged(CWnd* pFocusWnd)
{
    CListCtrl::OnPaletteChanged(pFocusWnd);

    if( pFocusWnd == this )
        return;

    OnQueryNewPalette();
}

```

(8) 从顶层窗口转发调色板消息

前面提到过, WM_QUERYNEWPALETTE 和 WM_PALETTECHANGED 消息是发送给顶层窗口的。因此当列表视图控件改变了调色板后,我们必须将这些消息发送回列表视图控件。下面假设应用程序为对话框,则相应的处理函数 OnQueryNewPalette 和 OnPaletteChanged 如清单 5-21 和 5-22 所示:

清单 5-21 OnPaletteChanged() 函数

```

void CListViewDlg::OnPaletteChanged(CWnd* pFocusWnd)
{
    CDialog::OnPaletteChanged(pFocusWnd);
    m_listctrl.SendMessage( WM_PALETTECHANGED, (LPARAM)pFocusWnd->m_hWnd );
}

```

清单 5-22 OnQueryNewPalette() 函数

```

BOOL CListViewDlg::OnQueryNewPalette()
{
    CDialog::OnQueryNewPalette();
    return m_listctrl.SendMessage( WM_QUERYNEWPALETTE );
}

```

5.6 改善列表视图控件的交互方式

一般来说,列表视图控件只是用于显示表格数据的。虽然 CListCtrl 类也支持一定程度的交互支持,但是这对于实际应用来说远远不够。本节将向读者介绍如何改善控件的交互方式。

5.6.1 在列表视图控件中使用复选框

列表视图控件本身就为用户界面设计提供了许多选择。其最优秀的特性之一就是能够在列中显示表数据、进行列排序、添加图像等等。而复选列表框通过每个条目上的复选框,使控件能够接收用户的输入。这两个控件分别由 CListCtrl 和 CCheckListBox 类提供支持。

如果能够将这两者结合起来,控件的功能不是更加强大吗?

一般来说,这时需要使用自绘制列表视图控件,并自己绘制复选框。不过在 Visual C++ 6.0 以上版本中无需如此,它为列表视图控件提供了一些新的状态标志,可以轻松地完成这一工作。设置可以通过 SetExtendedStyle 函数完成,示范代码如下:

```
SetExtendedStyle(LVS_EX_CHECKBOXES);
```

不过在设置了该选项后,就应该设法对复选状态的改变作出响应。这需要添加对 LVN_ITEMCHANGED 通告消息的处理函数,该通告消息在条目发生变化时发送。该通告消息的映射形式如下:

```
ON_NOTIFY(LVN_ITEMCHANGED, IDC_MYLIST, OnItemchangedLinksList)
```

其中 OnItemchangedLinksList 即为通告的处理函数,其示范形式如清单 5-23 所示:

清单 5-23 OnItemchangedLinksList() 函数

```
void DemoDlg::OnItemchangedLinksList(NMHDR* pNMHDR, LRESULT* pResult)
{
    NM_LISTVIEW* pNMListView = (NM_LISTVIEW*)pNMHDR;
    *pResult = 0;

    if (pNMListView->uOldState == 0 && pNMListView->uNewState == 0)
        return;    // 状态没有改变
    BOOL bPrevState = (BOOL)((((pNMListView->uOldState &
                                LVIS_STATEIMAGEMASK) >> 12) - 1)); // 旧的复选状态
    if (bPrevState < 0) // 在初始时,没有以前的状态
        bPrevState = 0; // 因此将其状态设置为 FALSE —— 未复选

    // 新的复选状态
    BOOL bChecked = (BOOL)((((pNMListView->uNewState & LVIS_STATEIMAGEMASK) >>
                                12) - 1));
    if (bChecked < 0) // 如果不是非复选通告,则将其设置为 FALSE
```



```
        bChecked = 0;

        if (bPrevState == bChecked) //复选框无变化
            return;

        // 现在 bChecked 变量即为新的复选框状态,以后可以根据其值进行处理
        // ....
    }
```

如果需要使用代码改变条目的复选状态,则可以使用如清单 5-24 所示的代码:

清单 5-24 SetLVCheck() 函数

```
void SetLVCheck (WPARAM ItemIndex, BOOL bCheck)
{
    ListView_SetItemState (m_lvTestList.m_hWnd, ItemIndex,
        UINT((int(bCheck) + 1) << 12), LVIS_STATEIMAGEMASK);
}
```

5.6.2 在位编辑子项

在默认情况下,列表视图控件具有可编辑属性时只有条目的第一项可以被编辑。这大大限制了这一属性的应用。因为并不是在所有的情况下,用户都只需改变列表视图中的第一项。如果不扩展这一属性,程序员在进行设计时可能要煞费苦心,以便使用户仅仅编辑第一项就行。

这一问题在一些数据库应用程序中尤其可能发生。许多能够以列表方式显示数据库记录的控件,例如:DataGrid、FlexGrid 等等,都必须首先为其绑定一个数据库,否则最多只能显示而不能编辑。读者可以想像,列表视图控件是一个非常方便的显示数据库记录的工具,如果能使其所有子项都能够被编辑,那么无疑能够解决这个问题。下面就向读者介绍如何为列表控件添加编辑子项的功能。

(1) 创建 CListCtrl 的派生类 CMyListCtrl

(2) 设计 HitTestEx 函数

HitTestEx 是 CListCtrl 类的 HitTest 成员函数的扩展。在此函数中除了确定单击位置所在条目的索引外,还要确定对应的列索引。这很容易理解,因为我们的目的就是使所有子项都能够被编辑,那么要定位将被编辑的子项,就必须同时得到其行、列索引。清单 5-25 所示为 HitTestEx 函数的源代码,其中 point 参数为鼠标单击位置,而 col 将返回单击处的列索引:

清单 5-25 HitTestEx() 函数

```
int CMyListCtrl::HitTestEx(CPoint &point, int *col) const
{
    int colnum = 0;
    int row = HitTest( point, NULL );

    if( col ) *col = 0;
```

```

// 确定列表视图控件为 LVS_REPORT 风格
if( (GetWindowLong(m_hWnd, GWL_STYLE) & LVS_TYPEMASK) != LVS_REPORT )
    return row;

// 得到最顶端和最底端的可见行
row = GetTopIndex();
int bottom = row + GetCountPerPage();
if( bottom > GetItemCount() )
    bottom = GetItemCount();

// 得到列数
CHeaderCtrl* pHeader = (CHeaderCtrl*)GetDlgItem(0);
int nColumnCount = pHeader->GetItemCount();

// 在可见行之间循环
for( ;row <= bottom;row++ )
{
    // 得到条目的边界矩形,并检查单击位置是否在该矩形中
    CRect rect;
    GetItemRect( row, &rect, LVIR_BOUNDS );
    if( rect.PtInRect(point) )
    {
        // 寻找单击位置所在的列索引
        for( colnum = 0; colnum < nColumnCount; colnum++ )
        {
            int colwidth = GetColumnWidth(colnum);
            if( point.x >= rect.left && point.x <= (rect.left + colwidth) )
            {
                if( col ) *col = colnum;
                return row;
            }
            rect.left += colwidth;
        }
    }
}
return -1;
}

```

(3) 添加编辑初始化函数 EditSubLabel

当单击已经被选中的行后,就可以开始对相应子项的编辑。在开始编辑操作前,需要进行一些预设置。EditSubLabel 函数使用行和列索引,并在创建编辑控件前使要编辑的子项可见。然后函数以合适的尺寸和文本调整创建编辑控件(该控件由 CInPlaceEdit 类管理,该类我们将在下面(7)中进行介绍)。清单 5-26 所示为 EditSubLabel 函数的源代码:

清单 5-26 EditSubLabel()函数

```

CEdit* CMyListCtrl::EditSubLabel( int nItem, int nCol )
{
    // 保证条目可见
    if( ! EnsureVisible( nItem, TRUE ) ) return NULL;
}

```

```

// 保证 nCol 合法
CHeaderCtrl * pHeader = (CHeaderCtrl *)GetDlgItem(0);
int nColumnCount = pHeader->GetItemCount();
if( nCol >= nColumnCount || GetColumnWidth(nCol) < 5 )
    return NULL;

// 得到列偏移量(以像素为单位)
int offset = 0;
for( int i = 0; i < nCol; i++ )
    offset += GetColumnWidth( i );

CRect rect;
GetItemRect( nItem, &rect, LVIR_BOUNDS );

// 如果列不可见,则滚动控件
CRect rcClient;
GetClientRect( &rcClient );
if( offset + rect.left < 0 || offset + rect.left > rcClient.right )
{
    CSize size;
    size.cx = offset + rect.left;
    size.cy = 0;
    Scroll( size );
    rect.left -= size.cx;
}

// 得到列的对齐方式
LV_COLUMN lvcol;
lvcol.mask = LVCF_FMT;
GetColumn( nCol, &lvcol );
DWORD dwStyle ;
if((lvcol.fmt&LVCFMT_JUSTIFYMASK) == LVCFMT_LEFT)
    dwStyle = ES_LEFT;
else if((lvcol.fmt&LVCFMT_JUSTIFYMASK) == LVCFMT_RIGHT)
    dwStyle = ES_RIGHT;
else dwStyle = ES_CENTER;

rect.left += offset + 4;
rect.right = rect.left + GetColumnWidth( nCol ) - 3 ;
if( rect.right > rcClient.right ) rect.right = rcClient.right;

dwStyle |= WS_BORDER|WS_CHILD|WS_VISIBLE|ES_AUTOHSCROLL;
CEdit * pEdit = new CInPlaceEdit(nItem, nCol, GetItemText( nItem, nCol ));
pEdit->Create( dwStyle, rect, this, IDC_IPEDIT );

return pEdit;
}

```

(4) 处理滚动消息

CInPlaceEdit 类负责在它失去输入焦点时,销毁编辑控件并删除对象。用户单击滚动条时,编辑控件显然应该失去输入焦点。因此需要添加对滚动消息的处理,在其中将强制编辑控件失去焦点,并将焦点转移到列表视图控件本身。清单 5-27 和 5-28 所示为滚动消

息处理函数:

清单 5-27 OnHScroll() 函数

```
void CMyListCtrl::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar * pScrollBar)
{
    if( GetFocus() != this ) SetFocus();
    CListCtrl::OnHScroll(nSBCode, nPos, pScrollBar);
}
```

清单 5-28 OnVScroll() 函数

```
void CMyListCtrl::OnVScroll(UINT nSBCode, UINT nPos, CScrollBar * pScrollBar)
{
    if( GetFocus() != this ) SetFocus();
    CListCtrl::OnVScroll(nSBCode, nPos, pScrollBar);
}
```

(5) 处理子项编辑

与默认的对第一列的编辑一样,对其他子项的编辑完成后,编辑控件也会发送 LVN_ENDLABELEDIT 通告消息。我们当然必须处理这一通告消息以便使编辑的结果被接受,并更新列表视图控件的显示。清单 5-29 所示为对 LVN_ENDLABELEDIT 通告消息的处理函数:

清单 5-29 OnEndLabelEdit() 函数

```
void CMyListCtrl::OnEndLabelEdit(NMHDR * pNMHDR, LRESULT * pResult)
{
    LV_DISPINFO * plvDispInfo = (LV_DISPINFO *)pNMHDR;
    LV_ITEM * plvItem = &plvDispInfo->item;

    if (plvItem->pszText != NULL)
    {
        SetItemText(plvItem->iItem, plvItem->iSubItem, plvItem->pszText);
    }
    *pResult = FALSE;
}
```

(6) 添加用户用以开始编辑的手段

一般来说,当用户单击已被选中的条目时,可以开始对被单击子项的编辑。当然也完全可以选择其他方式,但是正像我们在第 2 章中所指出的:用户界面必须遵守一定的规则,否则就会给用户带来不便。因此,本步的工作应该在对 WM_LBUTTONDOWN 消息的响应函数中进行。如果单击了已经被选中的条目,则为相应子项创建编辑控件。在下面的代码中,在创建编辑控件之前首先检查条目是否具有 LVS_EDITLABELS 风格。如果为条目的第一子项,则不激活编辑控件而让列表视图控件自己处理。OnLButtonDown 函数的源代码如清单 5-30 所示:

清单 5-30 OnLButtonDown() 函数

```
void CMyListCtrl::OnLButtonDown(UINT nFlags, CPoint point)
```

```

    {
        int index;
        CListCtrl::OnLButtonDown(nFlags, point);

        int colnum;
        if( ( index = HitTestEx( point, &colnum ) ) != -1 )
        {
            UINT flag = LVIS_FOCUSED;
            if( (GetItemState( index, flag ) & flag) == flag && colnum > 0)
            {
                // 检查 LVS_EDITLABELS 风格
                if( GetWindowLong(m_hWnd, GWL_STYLE) & LVS_EDITLABELS )
                    EditSubLabel( index, colnum );
            }
            else
                SetItemState( index, LVIS_SELECTED | LVIS_FOCUSED ,
                    LVIS_SELECTED | LVIS_FOCUSED );
        }
    }
}

```

(7) CInPlaceEdit 类

为了满足我们的特殊需要,必须扩展 CEdit 类的功能,亦即派生其子类(CInPlaceEdit)。其主要目的是使编辑结束时,类能够发送 LVN_ENDLABELEDIT 消息。并且当控件失去焦点,或用户按下 Escape 或 Enter 键时,销毁编辑控件并删除对象。清单 5-31 所示为类的头文件,其中声明了 4 个私有变量。这些变量将在发送 LVN_ENDLABELEDIT 消息时使用。

清单 5-31 CInPlaceEdit 类的头文件

```

// InPlaceEdit.h : header file
//

/////////////////////////////////////////////////////////////////
// CInPlaceEdit window

class CInPlaceEdit : public CEdit
{
// Construction
public:
    CInPlaceEdit(int iItem, int iSubItem, CString sInitText);

// Attributes
public:

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CInPlaceEdit)
    public:
        virtual BOOL PreTranslateMessage(MSG * pMsg);
    //}}AFX_VIRTUAL

```



```

// Implementation
public:
    virtual ~CInPlaceEdit();

    // Generated message map functions
protected:
   //{{AFX_MSG(CInPlaceEdit)
    afx_msg void OnKillFocus(CWnd* pNewWnd);
    afx_msg void OnNcDestroy();
    afx_msg void OnChar(UINT nChar, UINT nRepCnt, UINT nFlags);
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
   //}}AFX_MSG

    DECLARE_MESSAGE_MAP()
private:
    int m_iItem;
    int m_iSubItem;
    CString m_sInitText;
    BOOL m_bESC; // To indicate whether ESC key was pressed
};
/////////////////////////////////////////////////////////////////

```

CInPlaceEdit 类的构造函数简单地将传递给其的参数保存起来,并将 m_bESC 初始化为 FALSE。清单 5-32 所示为 CInPlaceEdit 类的构造、析构函数以及消息映射的源代码:

清单 5-32 CInPlaceEdit 类的头文件

```

CInPlaceEdit::CInPlaceEdit(int iItem, int iSubItem, CString sInitText)
:m_sInitText( sInitText )
{
    m_iItem = iItem;
    m_iSubItem = iSubItem;
    m_bESC = FALSE;
}

CInPlaceEdit::~~CInPlaceEdit()
{
}

BEGIN_MESSAGE_MAP(CInPlaceEdit, CEdit)
   //{{AFX_MSG_MAP(CInPlaceEdit)
    ON_WM_KILLFOCUS()
    ON_WM_NCDESTROY()
    ON_WM_CHAR()
    ON_WM_CREATE()
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()

```

而重载的 PreTranslateMessage 函数则确保指定的键对于编辑控件有效。Escape 和 Enter 键通常由 CDialog 或 CFormView 对象进行预翻译。因此在 PreTranslateMessage 函数中特

别检查这些键,并将其传递给编辑控件。GetKeyState(VK_CONTROL)函数确保了 Ctrl + C, Ctrl + V 和 Ctrl + X 对编辑控件有效。清单 5-33 所示为 PreTranslateMessage 函数的源代码:

清单 5-33 PreTranslateMessage() 函数

```

BOOL CInPlaceEdit::PreTranslateMessage(MSG * pMsg)
{
    if( pMsg->message == WM_KEYDOWN )
    {
        if(pMsg->wParam == VK_RETURN || pMsg->wParam == VK_DELETE
            || pMsg->wParam == VK_ESCAPE || GetKeyState( VK_CONTROL))
        {
            ::TranslateMessage(pMsg);
            ::DispatchMessage(pMsg);
            return TRUE;
        }
    }

    return CEdit::PreTranslateMessage(pMsg);
}

```

类中的 OnKillFocus 函数负责发送 LVN_ENDLABELEDIT 通告消息,并销毁编辑控件。实际上通告消息是发送给列表视图控件的父窗口,而不是控件本身。函数将根据 m_bESC 成员确定是否应该发送空字符串(亦即用户取消了编辑操作)。清单 5-34 所示为 OnKillFocus 函数的源代码:

清单 5-34 OnKillFocus() 函数

```

void CInPlaceEdit::OnKillFocus(CWnd * pNewWnd)
{
    CEdit::OnKillFocus(pNewWnd);

    CString str;
    GetWindowText(str);

    // 将通告发送给列表视图控件
    LV_DISPINFO dispinfo;
    dispinfo.hdr.hwndFrom = GetParent()->m_hWnd;
    dispinfo.hdr.idFrom = GetDlgCtrlID();
    dispinfo.hdr.code = LVN_ENDLABELEDIT;

    dispinfo.item.mask = LVIF_TEXT;
    dispinfo.item.iItem = m_iItem;
    dispinfo.item.iSubItem = m_iSubItem;
    dispinfo.item.pszText = m_bESC ? NULL : LPTSTR((LPCTSTR)str);
    dispinfo.item.cchTextMax = str.GetLength();

    GetParent()->GetParent()->SendMessage( WM_NOTIFY, GetParent()->GetDlgCtrlID(),(LPARAM)&dispinfo );

    DestroyWindow();
}

```

OnNcDestroy 函数负责销毁当前 C++ 对象,其源代码如清单 5-35 所示:

清单 5-35 OnNcDestroy() 函数

```
void CInPlaceEdit::OnNcDestroy()
{
    CEdit::OnNcDestroy();
    delete this;
}
```

OnChar 函数负责处理特殊的键盘输入:Escape 和 Enter 键。当检测到用户按下这两个键时将终止编辑,并将焦点还给列表视图控件,这会导致对 OnKillFocus 函数的调用。而其他键的输入则先由基类函数处理,然后才接着判断控件是否需要改变尺寸。首先使用正确的字体确定新字符串的长度,然后将其与编辑控件的当前尺寸相比较。如果编辑框不能完全显示字符串,则在确定父窗口(列表视图控件)中有空间后,改变编辑控件的尺寸。OnChar()函数的源代码如清单 5-36 所示:

清单 5-36 OnChar() 函数

```
void CInPlaceEdit::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    if( nChar == VK_ESCAPE || nChar == VK_RETURN)
    {
        if( nChar == VK_ESCAPE )
            m_bESC = TRUE;
        GetParent() -> SetFocus();
        return;
    }

    CEdit::OnChar(nChar, nRepCnt, nFlags);

    // 如果需要,则改变编辑控件的尺寸
    CString str;

    GetWindowText( str );
    CWindowDC dc(this);
    CFont * pFont = GetParent() -> GetFont();
    CFont * pFontDC = dc.SelectObject( pFont );
    CSize size = dc.GetTextExtent( str );
    dc.SelectObject( pFontDC );
    size.cx += 5;

    // 得到客户矩形
    CRect rect, parentrect;
    GetClientRect( &rect );
    GetParent() -> GetClientRect( &parentrect );

    // 将矩形转化为父窗口中的坐标
    ClientToScreen( &rect );
    GetParent() -> ScreenToClient( &rect );

    // 检查控件是否需要重新设置尺寸,亦即是否有可以增加的空间
```

```

    if( size.cx > rect.Width() )
    {
        if( size.cx + rect.left < parentrect.right )
            rect.right = rect.left + size.cx;
        else
            rect.right = parentrect.right;
        MoveWindow( &rect );
    }
}

```

OnCreate 函数则负责创建控件并初始化其中的内容,其源代码如清单 5-37 所示:

清单 5-37 OnCreate()函数

```

int CInPlaceEdit::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CEdit::OnCreate(lpCreateStruct) == -1)
        return -1;

    // 设置合适的字体
    CFont * font = GetParent()->GetFont();
    SetFont(font);

    SetWindowText( m_sInitText );
    SetFocus();
    SetSel( 0, -1 );
    return 0;
}

```

配套光盘的 chap6\lstviewedit 目录下为具有编辑功能的列表视图控件,不过它使用的是 ListView 派生类,但功能类似,请读者自行参阅。

5.6.3 使用组合框控件

在前面两节中,向读者介绍了如何允许用户编辑列表视图控件中的条目。在有些情况下,可能并不希望用户对条目作出太多改变,或希望能够给用户提供一些选项,而不是让用户自己进行编辑。这时使用组合框控件来取代编辑控件是一个非常好的方法。实现组合框控件的方法与步骤和实现编辑条目子项类似,具体步骤如下:

- (1) 创建 CListCtrl 的派生类 CMyListCtrl
- (2) 设计 HitTestEx 函数

HitTestEx 是 CListCtrl 类的 HitTest 成员函数的扩展。在此函数中除了确定单击位置所在条目的索引外,还要确定对应的列索引。因为我们的目的就是使所有子项都能够被编辑,而要定位将被编辑的子项,就必须同时得到其行、列索引。该函数的源代码参见清单 6-25。

- (3) 添加用以创建组合框控件的函数 ShowInPlaceList

该函数与清单 5-26 所示的 EditSubLabel 函数相似,其区别在于当函数结束时由 CIn-

PlaceList 类创建了一个组合框控件。需要注意的是,该函数也需要字符串列表作为参数,它作为用户可以选择的选项。清单 5-38 所示即为 ShowInPlaceList 函数的源代码,其中 nItem 为子项的行索引, nCol 为子项的列索引, lstItems 为组合框控件的选项, nSel 为组合框控件中的初始选项。

清单 5-38 ShowInPlaceList() 函数

```
CComboBox * CMyListCtrl:: ShowInPlaceList( int nItem, int nCol, CStringList
&lstItems, int nSel )
{
    // 确保子项可见
    if( ! EnsureVisible( nItem, TRUE ) ) return NULL;

    // 确保列索引有效
    CHeaderCtrl * pHeader = (CHeaderCtrl *)GetDlgItem(0);
    int nColumnCount = pHeader->GetItemCount();
    if( nCol >= nColumnCount || GetColumnWidth(nCol) < 10 )
        return NULL;

    // 得到列偏移
    int offset = 0;
    for( int i = 0; i < nCol; i++ )
        offset += GetColumnWidth( i );

    CRect rect;
    GetItemRect( nItem, &rect, LVIR_BOUNDS );

    // 滚动列表视图控件以使子项可见
    CRect rcClient;
    GetClientRect( &rcClient );
    if( offset + rect.left < 0 || offset + rect.left > rcClient.right )
    {
        CSize size;
        size.cx = offset + rect.left;
        size.cy = 0;
        Scroll( size );
        rect.left -= size.cx;
    }

    rect.left += offset + 4;
    rect.right = rect.left + GetColumnWidth( nCol ) - 3 ;
    int height = rect.bottom - rect.top;
    rect.bottom += 5 * height;
    if( rect.right > rcClient.right ) rect.right = rcClient.right;

    DWORD dwStyle =
    WS_BORDER|WS_CHILD|WS_VISIBLE|WS_VSCROLL|WS_HSCROLL
    |CBS_DROPDOWNLIST|CBS_DISABLENOSCROLL;
    CComboBox * pList = new CInPlaceList(nItem, nCol, &lstItems, nSel);
    pList->Create( dwStyle, rect, this, IDC_IPEDIT );
    pList->SetItemHeight( -1, height);
    pList->SetHorizontalExtent( GetColumnWidth( nCol ));
}
```



```
return pList;
```

(4) 处理滚动消息

CInPlaceList 类负责在它失去输入焦点时,销毁组合框控件并删除对象。单击滚动条时,组合框控件显然应该失去输入焦点。因此需要添加对滚动消息的处理,在其中将强制组合框控件失去焦点,并将焦点转移到列表视图控件本身。清单 5-39 和 5-40 所示为滚动消息处理函数:

清单 5-39 OnHScroll() 函数

```
void CMyListCtrl::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar * pScrollBar)
{
    if( GetFocus() != this ) SetFocus();
    CListCtrl::OnHScroll(nSBCode, nPos, pScrollBar);
}
```

清单 5-40 OnVScroll() 函数

```
void CMyListCtrl::OnVScroll(UINT nSBCode, UINT nPos, CScrollBar * pScrollBar)
{
    if( GetFocus() != this ) SetFocus();
    CListCtrl::OnVScroll(nSBCode, nPos, pScrollBar);
}
```

(5) 处理子项选择

与编辑控件相似,组合框控件也会在选择了某个条目后,发送 LVN_ENDLABELEDIT 通告消息。CInPlaceList 类当然必须处理这一通告消息以使编辑的结果被接受,并更新列表视图控件的显示。清单 5-41 所示为对 LVN_ENDLABELEDIT 通告消息的处理函数:

清单 5-41 OnEndLabelEdit() 函数

```
void CMyListCtrl::OnEndLabelEdit(NMHDR * pNMHDR, LRESULT * pResult)
{
    LV_DISPINFO * plvDispInfo = (LV_DISPINFO *)pNMHDR;
    LV_ITEM * plvItem = &plvDispInfo->item;

    if (plvItem->pszText != NULL)
    {
        SetItemText(plvItem->iItem, plvItem->iSubItem, plvItem->pszText);
    }

    *pResult = FALSE;
}
```

(6) 添加开始子项选择的手段

一般来说,当用户单击已被选中的条目时,就可以开始对被单击子项的编辑。因此,本步的工作应该在 WM_LBUTTONDOWN 消息的响应函数中进行。如果单击了已经被选中的条目,则为相应子项创建组合框控件。在下面的代码中,函数在创建控件之前首先检查条目是否具有 LVS_EDITLABELS 风格。OnLButtonDown 函数的源代码如清单 5-42 所示:

清单 5-42 OnLButtonDown() 函数

```

void CMyListCtrl::OnLButtonDown(UINT nFlags, CPoint point)
{
    int index;
    CListCtrl::OnLButtonDown(nFlags, point);

    int colnum;
    if( ( index = HitTestEx( point, &colnum ) ) != -1 )
    {
        UINT flag = LVIS_FOCUSED;
        if( (GetItemState( index, flag ) & flag) == flag )
        {
            // 检查控件是否具有 LVS_EDITLABELS 风格
            if( GetWindowLong(m_hWnd, GWL_STYLE) & LVS_EDITLABELS )
            {
                CStringList lstItems;
                lstItems.AddTail( "选项 1" );
                lstItems.AddTail( "选项 2" );
                lstItems.AddTail( "选项 3" );
                lstItems.AddTail( "选项 4" );
                lstItems.AddTail( "选项 5" );
                lstItems.AddTail( "选项 6" );
                ShowInPlaceList( index, colnum, lstItems, 2 );
            }
        }
        else
        {
            SetItemState( index, LVIS_SELECTED | LVIS_FOCUSED ,
                LVIS_SELECTED | LVIS_FOCUSED );
        }
    }
}

```

(7) CInPlaceList 类

为了满足预期的特殊要求,必须扩展 CComboBox 类的功能,亦即派生其子类(CInPlaceList)。其主要目的是使选择结束时,类能够发送 LVN_ENDLABELEDIT 消息,并且当控件失去焦点,或用户按下 Escape 或 Enter 键时,销毁组合框控件并删除对象。清单 5-43 所示为类的头文件,其中声明了 5 个私有变量,这些变量将在发送 LVN_ENDLABELEDIT 消息时使用。

清单 5-43 CInPlaceList 类的头文件

```

// InPlaceList.h : header file
//

/////////////////////////////////////////////////////////////////
// CInPlaceList window
class CInPlaceList : public CComboBox
{
// Construction
public:

```

```

        CInPlaceList(int iItem, int iSubItem, CStringList *plstItems, int nSel);

// Attributes
public:

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CInPlaceList)
    public:
        virtual BOOL PreTranslateMessage(MSG * pMsg);
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CInPlaceList();

    // Generated message map functions
protected:
    //{{AFX_MSG(CInPlaceList)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnKillFocus(CWnd * pNewWnd);
    afx_msg void OnChar(UINT nChar, UINT nRepCnt, UINT nFlags);
    afx_msg void OnNcDestroy();
    afx_msg void OnCloseup();
    //}}AFX_MSG

    DECLARE_MESSAGE_MAP()
private:
    int m_iItem;
    int m_iSubItem;
    CStringList m_lstItems;
    int m_nSel;
    BOOL m_bESC;
};

```

CInPlaceList 类的构造函数简单地将传递给其的参数保存起来,并将 m_bESC 初始化为 FALSE。清单 5-44 所示为 CInPlaceList 类的构造、析构函数以及消息映射的源代码:

清单 5-44 CInPlaceList 类的头文件

```

CInPlaceList::CInPlaceList(int iItem, int iSubItem, CStringList *plstItems,
int nSel)
{
    m_iItem = iItem;
    m_iSubItem = iSubItem;

    m_lstItems.AddTail(plstItems);
    m_nSel = nSel;
    m_bESC = FALSE;
}

```

```

}

CInPlaceList::CInPlaceList()
{
}

BEGIN_MESSAGE_MAP(CInPlaceList, CComboBox)
    //{{AFX_MSG_MAP(CInPlaceList)
    ON_WM_CREATE()
    ON_WM_KILLFOCUS()
    ON_WM_CHAR()
    ON_WM_NCDESTROY()
    ON_CONTROL_REFLECT(CBN_CLOSEUP, OnCloseup)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

```

OnCreate 函数则负责创建控件并初始化其中的内容,其源代码如清单 5-45 所示:

清单 5-45 PreTranslateMessage() 函数

```

int CInPlaceList::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CComboBox::OnCreate(lpCreateStruct) == -1)
        return -1;

    // 设置合适的字体
    CFont * font = GetParent() -> GetFont();
    SetFont(font);

    for( POSITION pos = m_lstItems.GetHeadPosition(); pos != NULL; )
    {
        AddString( (LPCTSTR) (m_lstItems.GetNext( pos )) );
    }
    SetCurSel( m_nSel );
    SetFocus();
    return 0;
}

```

而重载的 PreTranslateMessage 函数则确保指定的键对于组合框有效。Escape 和 Enter 键通常由 CDialog 或 CformView 对象进行预翻译。因此在 PreTranslateMessage 函数中特别检查这些键,并将其传递给组合框控件。清单 5-46 所示为 PreTranslateMessage 函数的源代码:

清单 5-46 PreTranslateMessage() 函数

```

BOOL CInPlaceList::PreTranslateMessage(MSG * pMsg)
{
    if( pMsg -> message == WM_KEYDOWN )
    {
        if(pMsg -> wParam == VK_RETURN || pMsg -> wParam == VK_ESCAPE)
        {
            ::TranslateMessage(pMsg);
        }
    }
}

```

```

        ::DispatchMessage(pMsg);
        return TRUE;
    }
}
return CComboBox::PreTranslateMessage(pMsg);
}

```

类中的 `OnKillFocus` 函数负责发送 `LVN_ENDLABELEDIT` 通告消息,并销毁组合框控件。实际上通告消息是发送给列表视图控件的父窗口,而不是控件本身。发送通告消息时,将根据 `m_bESC` 成员确定是否应该发送空字符串(亦即用户取消了选择操作)。清单 5-47 所示为 `OnKillFocus` 函数的源代码:

清单 5-47 `OnKillFocus()` 函数

```

void CInPlaceList::OnKillFocus(CWnd* pNewWnd)
{
    CComboBox::OnKillFocus(pNewWnd);

    CString str;
    GetWindowText(str);

    // 将通告消息发送给列表视图控件的父窗口
    LV_DISPINFO dispinfo;
    dispinfo.hdr.hwndFrom = GetParent() -> m_hWnd;
    dispinfo.hdr.idFrom = GetDlgCtrlID();
    dispinfo.hdr.code = LVN_ENDLABELEDIT;

    dispinfo.item.mask = LVIF_TEXT;
    dispinfo.item.iItem = m_iItem;
    dispinfo.item.iSubItem = m_iSubItem;
    dispinfo.item.pszText = m_bESC ? NULL : LPTSTR((LPCTSTR)str);
    dispinfo.item.cchTextMax = str.GetLength();

    GetParent() -> GetParent() -> SendMessage( WM_NOTIFY, GetParent() -> GetDlgCtrlID(), (LPARAM)&dispinfo );
    PostMessage( WM_CLOSE );
}

```

`OnNcDestroy` 函数负责销毁当前 C++ 对象,其源代码如清单 5-48 所示:

清单 5-48 `OnNcDestroy()` 函数

```

void CInPlaceList::OnNcDestroy()
{
    CComboBox::OnNcDestroy();
    delete this;
}

```

`OnChar` 函数负责处理特殊的键盘输入:Escape 和 Enter 键。当检测到用户按下这两个键时将终止选择,并将焦点还给列表视图控件,这会导致对 `OnKillFocus` 函数的调用。而其他键的输入则先由基类函数处理,然后才接着判断控件是否需要改变尺寸。`OnChar()` 函数的源代码如清单 5-49 所示:

清单 5-49 OnChar() 函数

```
void CInPlaceList::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    if( nChar == VK_ESCAPE || nChar == VK_RETURN)
    {
        if( nChar == VK_ESCAPE )
            m_bESC = TRUE;
        GetParent() -> SetFocus();
        return;
    }

    CComboBox::OnChar(nChar, nRepCnt, nFlags);
}
```

当用户从组合框中选择了一项后 CInPlaceList 类将调用 OnCloseup 函数,以将输入焦点还给列表视图控件,并终止当前选择。清单 5-50 所示为 OnCloseup 函数的源代码:

清单 5-50 OnCloseup() 函数

```
void CInPlaceList::OnCloseup()
{
    GetParent() -> SetFocus();
}
```

5.6.4 增强子项在位编辑性能

上面一节介绍了如何编辑子项,本节将介绍另外一种实现子项编辑的方法,另外还介绍如何使用多行编辑控件。本节还将介绍多种进入编辑状态的手段,以及在同一控件中实现多种编辑方式的方法。图 5-4 所示即为具有多行编辑功能的列表视图控件。

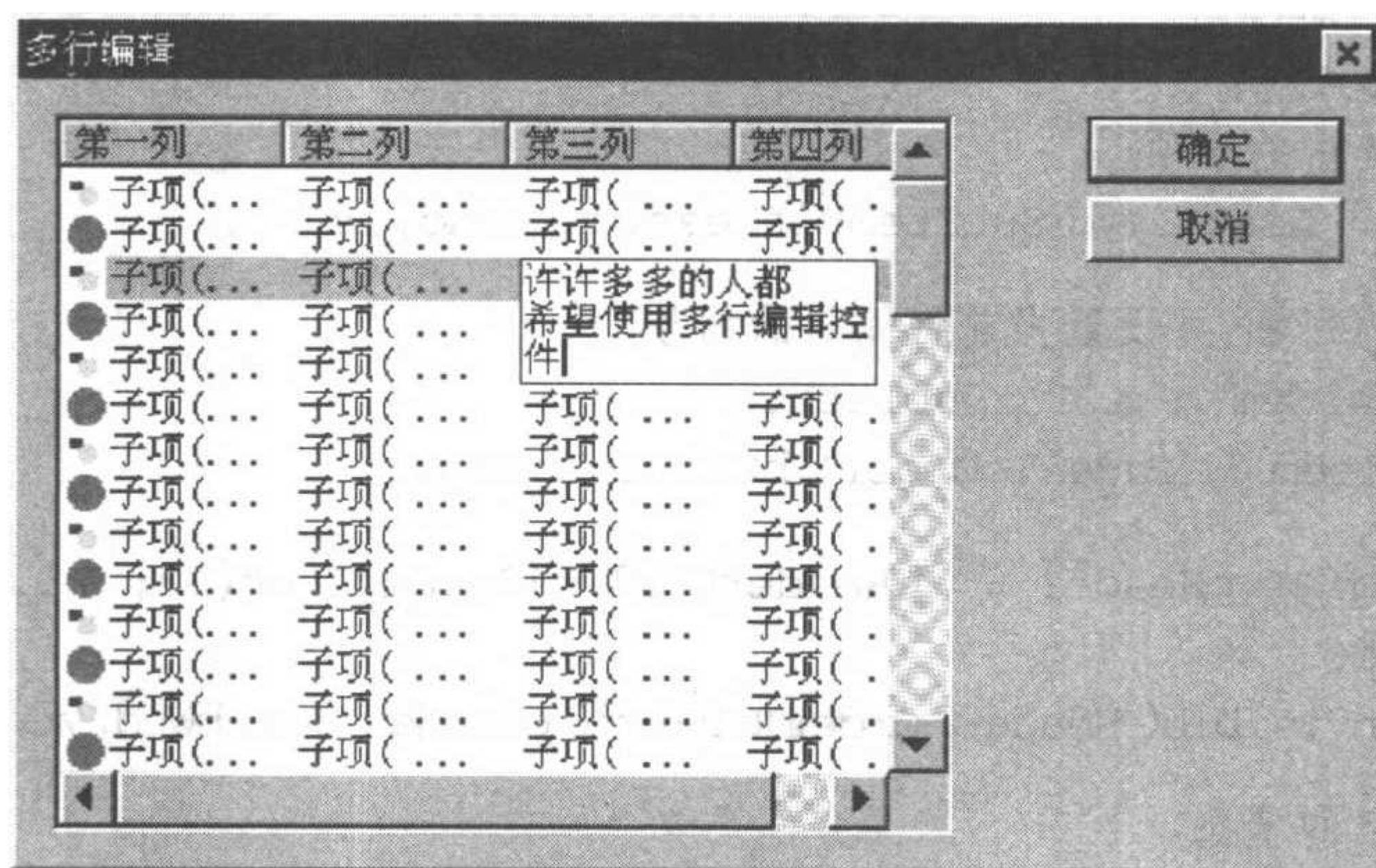


图 5-4 具有多行编辑功能的列表视图控件

下面介绍实现步骤。

注意 在编辑控件中如果希望插入回车,而又不想终止当前编辑,则应该按下 Ctrl-Enter 键。

(1) 创建 CListCtrl 的派生类 CGridListCtrl

将类名定为 CGridListCtrl 是因为它还能实现在列表视图中绘制网格线的功能。绘制网格线的实现非常简单,只需要重载类的 OnPaint 函数,并在其中选择列和行的分界位置绘制直线即可(实际上,绘制水平网格线的功能只需要为列表控件指定 LVS_EX_GRIDLINES 风格即可)。另外,读者需要注意一点,如果某种绘制是与单个条目相关(例如,改变条目被选择时的背景色),那么一般需要在 DrawItem 中进行;而如果绘制是与整个列表视图控件相关,则需要重载 OnPaint 函数。类需要添加一个成员变量,以保存当前用户正在操作的列索引。

```
// Attributes
public:
    //当前操作的列索引
    int m_CurSubItem;
```

(2) 响应鼠标单击以开始编辑

在用户界面设计的原则中已经向读者介绍过,对于同一操作应该为用户提供尽可能多的完成方式,例如既可以通过鼠标完成也可以通过键盘完成。如果用户使用鼠标,则首先需要确定被单击的子项。如何完成这一点上面一节已经进行了介绍,不过那是我们自己设计代码完成的,实际上使用 MFC 新添加的宏 ListView_SubItemHitTest 就能够很轻易地完成。下面我们还将使用 Header_OrderToIndex 宏以得到被单击的子项的排序(这与索引不一定一样,例如在排序后)。当确定了将编辑的子项后就可以向父窗口发送开始编辑的消息。清单 5-51 所示即为鼠标左键单击的消息响应函数 OnLButtonDown 的源代码:

清单 5-51 OnLButtonDown() 函数

```
void CGridListCtrl::OnLButtonDown(UINT nFlags, CPoint point)
{
    LVHITTESTINFO ht;
    ht.pt = point;
    // 确定被单击的子项
    int rval = ListView_SubItemHitTest( m_hWnd, &ht );

    // 存储旧的列号,并设置当前将编辑的列号
    int oldsubitem = m_CurSubItem;
    m_CurSubItem = IndexToOrder( ht.iSubItem );

    CHeaderCtrl* pHeader = (CHeaderCtrl*)GetDlgItem(0);
    // 使列可见
    MakeColumnVisible( Header_OrderToIndex( pHeader->m_hWnd, m_CurSubItem ) );

    // 存储条目的状态
    int state = GetItemState( ht.iItem, LVIS_FOCUSED );

    // 调用默认的鼠标左键消息响应函数
    CListCtrl::OnLButtonDown(nFlags, point);

    // 判断条目以前是否被选中,以及用户是否已经单击过当前子项一次
    if( ! state || m_CurSubItem == -1 || oldsubitem != m_CurSubItem )
```

```

        return;

    int doedit = 0;
    // 如果单击的为第 0 列,则确定用户是否单击了条目标签
    if( 0 == ht.iSubItem )
    {
        if( ht.flags & LVHT_ONITEMLABEL ) doedit = 1;
    }
    else
    {
        doedit = 1;
    }
    if( ! doedit ) return;

    // 向控件的父窗口发送通告消息
    CString str;
    str =.GetItemText( ht.iItem, ht.iSubItem );
    LV_DISPINFO dispinfo;
    dispinfo.hdr.hwndFrom = m_hWnd;
    dispinfo.hdr.idFrom = GetDlgCtrlID();
    dispinfo.hdr.code = LVN_BEGINLABELEDIT;

    dispinfo.item.mask = LVIF_TEXT;
    dispinfo.item.iItem = ht.iItem;
    dispinfo.item.iSubItem = ht.iSubItem;
    dispinfo.item.pszText = (LPTSTR)((LPCTSTR)str);
    dispinfo.item.cchTextMax = str.GetLength();

    GetParent() -> SendMessage( WM_NOTIFY, GetDlgCtrlID(),
        (LPARAM)&dispinfo );
}

int CGridListCtrl::IndexToOrder( int iIndex )
{
    // 由于控件只提供了 OrderToIndex 宏,
    // 因此我们必须自己设计如何将列索引转化为列顺序
    CHeaderCtrl * pHeader = (CHeaderCtrl *)GetDlgItem(0);
    int colcount = pHeader -> GetItemCount();
    int * orderarray = new int[ colcount ];
    Header_GetOrderArray( pHeader -> m_hWnd, colcount, orderarray );
    int i;
    for( i = 0; i < colcount; i++ )
    {
        if( orderarray[i] == iIndex )
            return i;
    }
    return -1;
}

```

用户同时还应该能够使用键盘开始编辑。重载了的 `PreTranslateMessage` 函数使用户按下 F2 键时就能够开始编辑。此外,还需要添加对键盘移动选择的支持。清单 5-52 所示为 `PreTranslateMessage` 函数的源代码:

清单 5-52 PreTranslateMessage() 函数

```

BOOL CGridListCtrl::PreTranslateMessage(MSG * pMsg)
{
    if(pMsg->message == WM_KEYDOWN)
    {
        // 处理键盘移动选择
        if( this == GetFocus() )
        {
            switch( pMsg->wParam )
            {
                case VK_LEFT:
                {
                    // 减少列号
                    m_CurSubItem--;
                    if( m_CurSubItem < -1 )
                    {
                        // 整行都被选中,而 F2 无意义
                        m_CurSubItem = -1;
                    }
                }
                else
                {
                    CHeaderCtrl * pHeader = (CHeaderCtrl *)GetDlgItem(0);
                    MakeColumnVisible( Header_OrderToIndex( pHeader->m_hWnd,
                                                                m_CurSubItem ) );

                    int iItem = GetNextItem( -1, LVNI_FOCUSED );
                    if( iItem != -1 )
                    {
                        CRect rcBounds;
                        GetItemRect(iItem, rcBounds, LVIR_BOUNDS);
                        InvalidateRect( &rcBounds );
                    }
                }
            }
            return TRUE;
        }
        case VK_RIGHT:
        {
            // 增加列号
            m_CurSubItem++;
            CHeaderCtrl * pHeader = (CHeaderCtrl *)GetDlgItem(0);
            int nColumnCount = pHeader->GetItemCount();
            // 不能超过最后一列
            if( m_CurSubItem > nColumnCount - 1 )
            {
                m_CurSubItem = nColumnCount - 1;
            }
        }
        else
        {
            MakeColumnVisible( Header_OrderToIndex( pHeader->m_

```

```

        hWnd, m_CurSubItem ) );
        int iItem = GetNextItem( -1, LVNI_FOCUSED );
        if( iItem != -1 )
        {
            CRect rcBounds;
            GetItemRect(iItem, rcBounds, LVIR_BOUNDS);
            InvalidateRect( &rcBounds );
        }
    }
    return TRUE;
case VK_F2: // 进入编辑模式
    {
        int iItem = GetNextItem( -1, LVNI_FOCUSED );
        if( m_CurSubItem != -1 && iItem != -1 )
        {
            // 向列表视图控件的父窗口发送通告消息
            CHeaderCtrl* pHeader = (CHeaderCtrl*)GetDlgItem(0);
            CString str;

            str = GetItemText( iItem, Header_OrderToIndex( pHeader->m_
                hWnd, m_CurSubItem ) );
            LV_DISPINFO dispinfo;
            dispinfo.hdr.hwndFrom = m_hWnd;
            dispinfo.hdr.idFrom = GetDlgCtrlID();
            dispinfo.hdr.code = LVN_BEGINLABELEDIT;

            dispinfo.item.mask = LVIF_TEXT;
            dispinfo.item.iItem = iItem;

            dispinfo.item.iSubItem = Header_OrderToIndex( pHeader->m_
                hWnd, m_CurSubItem );
            dispinfo.item.pszText = (LPTSTR)((LPCTSTR)str);
            dispinfo.item.cchTextMax = str.GetLength();
            GetParent( ) -> SendMessage( WM_NOTIFY, GetDlgCtrlID( ),
                (LPARAM)&dispinfo );
        }
    }
    break;
default:
    break;
}
}

return CListCtrl::PreTranslateMessage(pMsg);
}

```

(3) 使列可见

如果用户单击的子项所在的列并不完全可见,那么必须将其完全显示。下面将添加的函数就要完成这一功能。清单 5-53 所示为 MakeColumnVisible 函数的源代码:

清单 5-53 MakeColumnVisible() 函数

```

void CGridListCtrl::MakeColumnVisible(int nCol)
{
    if( nCol < 0 )
        return;

    // 得到列号数组,以得到列偏移
    CHeaderCtrl * pHeader = (CHeaderCtrl *)GetDlgItem(0);
    int colcount = pHeader->GetItemCount();
    ASSERT( nCol < colcount );
    int * orderarray = new int[ colcount ];
    Header_GetOrderArray( pHeader->m_hWnd, colcount, orderarray );
    // 得到列偏移
    int offset = 0;
    for( int i = 0; orderarray[i] != nCol; i++ )
        offset += GetColumnWidth( orderarray[i] );
    int colwidth = GetColumnWidth( nCol );
    delete[] orderarray;

    CRect rect;
    GetItemRect( 0, &rect, LVIR_BOUNDS );

    // 现在滚动控件,以使列可见
    CRect rcClient;
    GetClientRect( &rcClient );
    if( offset + rect.left < 0 || offset + colwidth + rect.left > rcClient.
        right )
    {
        CSize size;
        size.cx = offset + rect.left;
        size.cy = 0;
        Scroll( size );
        rect.left -= size.cx;
    }
}

```

(4) 处理滚动消息

CInPlaceEdit 类负责在它失去输入焦点时,销毁编辑控件并删除对象。单击滚动条时,编辑控件显然应该失去输入焦点。因此需要添加对滚动消息的处理,并在其中将强制编辑控件失去焦点,然后将焦点转移到列表视图控件本身。对滚动消息的响应函数参见清单 5-27 和 5-28。

(5) 确定编辑控件的出现位置

读者可能注意到列表视图控件并不处理编辑结束和开始的消息(这与本节的前两种实现方式不同)。之所以让列表视图控件的父窗口处理编辑控件实例的创建和析构,原因在于该操作由列表视图需要很长时间完成,而由其父窗口创建一次即可并能够重复使用。既然由于编辑控件并不是由列表视图控件完成的,那么其定位就需要由列表控件进行。清单 5-54 所示的 PositionControl 函数即可以完成此工作:

清单 5-54 PositionControl() 函数

```

BOOL CGridListCtrl::PositionControl( CWnd * pWnd, int iItem, int iSubItem )
{
    ASSERT( pWnd && pWnd->m_hWnd );
    ASSERT( iItem >= 0 );
    // 使子项可见
    if( ! EnsureVisible( iItem, TRUE ) ) return NULL;

    // 使 nCol is 有效
    CHeaderCtrl* pHeader = (CHeaderCtrl*)GetDlgItem(0);
    int nColumnCount = pHeader->GetItemCount();
    ASSERT( iSubItem >= 0 && iSubItem < nColumnCount );
    if( iSubItem >= nColumnCount ||
        GetColumnWidth(Header_OrderToIndex( pHeader->m_hWnd,iSubItem)) < 5 )
    {
        return 0;
    }

    int *orderarray = new int[ nColumnCount ];
    Header_GetOrderArray( pHeader->m_hWnd, nColumnCount, orderarray );
    int offset = 0;
    int i;
    for( i = 0; orderarray[i] != iSubItem; i++ )
        offset += GetColumnWidth( orderarray[i] );
    int colwidth = GetColumnWidth( iSubItem );
    delete[] orderarray;

    CRect rect;
    GetItemRect( iItem, &rect, LVIR_BOUNDS );

    // 滚动控件使列可见
    CRect rcClient;
    GetClientRect( &rcClient );
    if( offset + rect.left < 0 || offset + colwidth + rect.left > rcClient.right )
    {
        CSize size;
        size.cx = offset + rect.left;
        size.cy = 0;
        Scroll( size );
        rect.left -= size.cx;
    }

    rect.left += offset + 4;
    rect.right = rect.left + colwidth - 3 ;
    // 编辑控件的右边界不应该超过列表视图控件的右边界
    if( rect.right > rcClient.right )
        rect.right = rcClient.right;
    pWnd->MoveWindow( &rect );

    return 1;
}

```

(6) CInPlaceEdit 类

为了满足我们的特殊需求,必须扩展 CEdit 类的功能,亦即派生其子类(CInPlaceEdit)。其主要目的是使编辑结束时,类能够发送 LVN_ENDLABELEDIT 消息。并且当控件失去焦点,或用户按下 Escape 或 Enter 键时,销毁编辑控件并删除对象。清单 5-55 所示为类的头文件,其中声明了 4 个私有变量。这些变量将在发送 LVN_ENDLABELEDIT 消息时使用。本节定义的 CInPlaceEdit 类与 5.3 节中定义的 CInPlaceEdit 类基本相同,请读者参照 5.1 节中的介绍以及 chap6/gridlist 目录下的源代码。

(7) 处理编辑开始和结束消息

前面已经提到过,对子项编辑消息的响应是由列表视图控件的父窗口负责处理的。列表视图控件的父窗口必须创建编辑控件,并当编辑结束时销毁它。其最大的方便之处就是无需显式销毁编辑控件对象,而是当窗口本身被销毁时编辑控件对象也会被销毁。这使得编辑控件能够被重新使用。这种编辑子项的方法,允许应用程序决定编辑控制的风格:单行编辑控件、多行编辑控件或是组合框控件。假设列表视图控件的父窗口为 TestDlg,则相应的消息处理函数如清单 5-55 和 5-56 所示:

清单 5-55 OnBeginlabeleditList() 函数

```
void TestDlg::OnBeginlabeleditList(NMHDR * pNMHDR, LRESULT * pResult)
{
    LV_DISPINFO * pDispInfo = (LV_DISPINFO *)pNMHDR;

    CString str = pDispInfo->item.pszText;
    int item = pDispInfo->item.iItem;
    int subitem = pDispInfo->item.iSubItem;

    // 构造并创建多行编辑控件,当然同时我们也可以选择组合框、复选框等其他控件
    m_pListEdit = new CInPlaceEdit( item, subitem, str );
    // Start with a small rectangle. We'll change it later.
    CRect rect( 0,0,1,1 );
    DWORD dwStyle = ES_LEFT;
    dwStyle |= WS_BORDER | WS_CHILD | WS_VISIBLE | ES_MULTILINE | ES_AUTOVSCROLL;
    m_pListEdit->Create( dwStyle, rect, &m_GridListCtrl, 103 );
    // 得到编辑控件应在的位置
    m_GridListCtrl.PositionControl( m_pListEdit, item, subitem );
    // 使编辑控件的尺寸与内容相适合
    m_pListEdit->CalculateSize();
    // 返回 TRUE,这样使列表控件不能自己处理编辑操作
    *pResult = TRUE;
}
```

清单 5-56 OnEndlabeleditList() 函数

```
void TestDlg::OnEndlabeleditList(NMHDR * pNMHDR, LRESULT * pResult)
{
    LV_DISPINFO * pDispInfo = (LV_DISPINFO *)pNMHDR;
    int item = pDispInfo->item.iItem;
    int subitem = pDispInfo->item.iSubItem;
    if( m_pListEdit )
```

```

    {
        CString str;
        if( pDispInfo->item.pszText )
            m_GridListCtrl.SetItemText( item, subitem, pDispInfo->item.psz-
                Text );

        delete m_pListEdit;
        m_pListEdit = 0;
    }
    *pResult = 0;
}

```

5.6.5 内容提示

内容提示与工具提示相似,对于列表视图控件来说,就是当鼠标移动到某子项上方时,内容提示中就显示子项内容。这在列宽不够,无法显示所有子项内容时尤其有用。图 5-5 所示即为具有内容提示的列表视图控件:

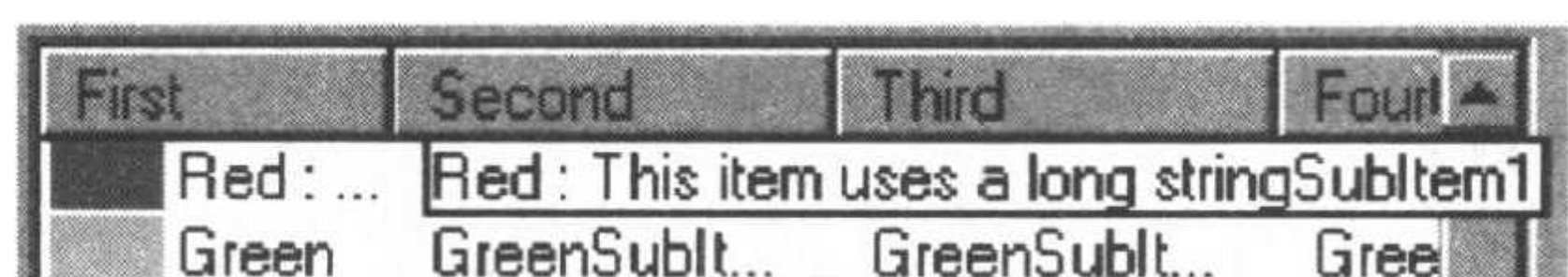


图 5-5 具有内容提示的列表视图控件

本节将创建一个定制类 CTitleTip 用以管理内容提示。这样,只要在鼠标移动消息处理函数中简单地调用 CTitleTip 对象的成员函数即可。此后 CTitleTip 对象将决定是否显示内容提示。当鼠标移出当前子项的边界后,或应用程序失去焦点时内容提示就会销毁。下面将向读者介绍其实现步骤。

(1) 创建 CTitleTip 类

CTitleTip 类的定义如清单 5-57 所示。这是一个非常通用的类,它能与多种不同的控件一起使用。

清单 5-57 CTitleTip 类的定义

```

#define TITLETIP_CLASSNAME _T("ZTitleTip")

class CTitleTip : public CWnd
{
// Construction
public:
    CTitleTip();

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CTitleTip)
public:
    virtual BOOL PreTranslateMessage(MSG* pMsg);
    //}}AFX_VIRTUAL

```

```

// Implementation
public:
    void Show( CRect rectTitle, LPCTSTR lpszTitleText, int xoffset = 0);
    virtual BOOL Create( CWnd * pParentWnd);
    virtual ~CTitleTip();

protected:
    CWnd * m_pParentWnd;
    CRect m_rectTitle;

// Generated message map functions
protected:
    ///||AFX_MSG(CTitleTip)
    afx_msg void OnMouseMove(UINT nFlags, CPoint point);
    ///||AFX_MSG
    DECLARE_MESSAGE_MAP()
};

```

如果 CTitleTip 类还没有被程序的其他实例注册,则将在其构造函数中完成注册,并且类使用的背景画笔为 COLOR_INFOBK,这与工具提示所使用的颜色一致。清单 5-58 所示为 CTitleTip 类的构造函数:

清单 5-58 CTitleTip()函数

```

CTitleTip::CTitleTip()
{
    // 注册 Windows 类
    WNDCLASS wndcls;
    HINSTANCE hInst = AfxGetInstanceHandle();
    if(! (::GetClassInfo(hInst, TITLETIP_CLASSNAME, &wndcls)))
    {
        // 否则应该注册一个新类
        wndcls.style = CS_SAVEBITS ;
        wndcls.lpfnWndProc = ::DefWindowProc;
        wndcls.cbClsExtra = wndcls.cbWndExtra = 0;
        wndcls.hInstance = hInst;
        wndcls.hIcon = NULL;
        wndcls.hCursor = LoadCursor( hInst, IDC_ARROW );
        wndcls.hbrBackground = (HBRUSH)(COLOR_INFOBK + 1);
        wndcls.lpszMenuName = NULL;
        wndcls.lpszClassName = TITLETIP_CLASSNAME;
        if (! AfxRegisterClass(&wndcls))
            AfxThrowResourceException();
    }
}

```

Create 函数的功能与其他 MFC 类的 Create()函数类似,即创建窗口。此函数中读者最需要注意的一点为窗口的风格。指定 WS_BORDER 风格将使内容提示窗口四周具有边界。而 WS_POPUP 风格则允许内容提示窗口出现在列表视图控件的边界上。如果不指定 WS_POPUP 风格,则若内容提示窗口出现在控件窗口边界上,它就会被边界裁剪,而这

显然是我们所不希望出现的。指定了 `WS_EX_TOOLWINDOW` 风格使内容提示不会出现在任务条上。而 `WS_EX_TOPMOST` 风格则确保了内容提示可见。清单 5-59 所示为 `Create` 函数的源代码：

清单 5-59 `Create()` 函数

```

BOOL CTitleTip::Create(CWnd * pParentWnd)
{
    ASSERT_VALID(pParentWnd);

    DWORD dwStyle = WS_BORDER | WS_POPUP;
    DWORD dwExStyle = WS_EX_TOOLWINDOW | WS_EX_TOPMOST;
    m_pParentWnd = pParentWnd;
    return CreateEx( dwExStyle, TITLETIP_CLASSNAME, NULL, dwStyle, 0, 0, 0, 0,
        NULL, NULL, NULL );
}

```

`Show` 函数将被客户控件(这里为列表视图控件)不断调用。该函数的基本功能就是确定子项文本是否完全显示,如果不是则显示内容提示。同时它还存储内容提示的窗口矩形,以便当鼠标离开该子项时移除内容提示。清单 5-60 所示为 `Show` 函数的源代码,其中 `rectTitle` 为内容提示窗口矩形, `lpszTitleText` 为将显示的文本, `xoffset` 为窗口距离子项左边界的距离。

清单 5-60 `Show()` 函数

```

void CTitleTip::Show(CRect rectTitle, LPCTSTR lpszTitleText, int xoffset /* = 0 */)
{
    ASSERT( ::IsWindow( m_hWnd ) );
    ASSERT( ! rectTitle.IsRectEmpty() );

    // 如果已经显示了内容提示,则不作任何事
    if( IsWindowVisible() )
        return;

    // 如果应用程序没有焦点,则不显示内容提示
    if( GetFocus() == NULL )
        return;

    // 确定了内容提示的边界窗口
    m_rectTitle.top = -1;
    m_rectTitle.left = -xoffset - 1;
    m_rectTitle.right = rectTitle.Width() - xoffset;
    m_rectTitle.bottom = rectTitle.Height() + 1;

    // 得到文本的宽度
    m_pParentWnd->ClientToScreen( rectTitle );

    CClientDC dc(this);
    CString strTitle(lpszTitleText);
    CFont * pFont = m_pParentWnd->GetFont(); // use same font as ctrl
    CFont * pFontDC = dc.SelectObject( pFont );
}

```

```

CRect rectDisplay = rectTitle;
CSize size = dc.GetTextExtent( strTitle );
rectDisplay.left += xoffset;
rectDisplay.right = rectDisplay.left + size.cx + 3;

// 如果文本显示完全,则不显示内容提示
if( rectDisplay.right <= rectTitle.right - xoffset )
    return;

// 显示内容提示
SetWindowPos( &wndTop, rectDisplay.left, rectDisplay.top, rectDisplay.
    Width(), rectDisplay.Height(), SWP_SHOWWINDOW|SWP_NOACTIVATE );

dc.SetBkMode( TRANSPARENT );
dc.TextOut( 0, 0, strTitle );
dc.SelectObject( pFontDC );

SetCapture();
}

```

WM_MOUSEMOVE 消息的处理函数 OnMouseMove 负责检查鼠标所在位置是否在与内容提示相关的子项矩形中。显然子项矩形应该小于内容提示矩形,否则根本没有必要显示内容提示。如果鼠标在该矩形外,则隐藏先前的内容提示,并将 WM_MOUSEMOVE 或 WM_NCMOUSEMOVE 消息传递给底层窗口。清单 5-61 所示为 OnMouseMove 函数的源代码:

清单 5-61 OnMouseMove() 函数

```

void CTitleTip::OnMouseMove(UINT nFlags, CPoint point)
{
    if (! m_rectTitle.PtInRect(point)) {
        ReleaseCapture();
        ShowWindow( SW_HIDE );

        ClientToScreen( &point );
        CWnd * pWnd = WindowFromPoint( point );
        if ( pWnd == this )
            pWnd = m_pParentWnd;
        int hittest = (int)pWnd->SendMessage(WM_NCHITTEST,
            0,MAKELONG(point.x,point.y));
        if (hittest == HTCLIENT) {
            pWnd->ScreenToClient( &point );
            pWnd->PostMessage( WM_MOUSEMOVE, nFlags,
                MAKELONG(point.x,point.y) );
        } else {
            pWnd->PostMessage( WM_NCMOUSEMOVE, hittest,
                MAKELONG(point.x,point.y) );
        }
    }
}

```

当用户按下任意键或鼠标按钮后,内容提示应该消失。因此我们重载 PreTrans-

lateMessage 函数以检查这些消息。如果接收到这些消息,则内容提示将被移除并将合适的消息传递给列表视图控件。清单 5-62 所示为 PreTranslateMessage 函数的源代码:

清单 5-62 PreTranslateMessage() 函数

```

BOOL CTitleTip::PreTranslateMessage(MSG * pMsg)
{
    CWnd * pWnd;
    int hittest;
    switch( pMsg->message )
    {
    case WM_LBUTTONDOWN:
    case WM_RBUTTONDOWN:
    case WM_MBUTTONDOWN:
        POINTS pts = MAKEPOINTS( pMsg->lParam );
        POINT point;
        point.x = pts.x;
        point.y = pts.y;
        ClientToScreen( &point );
        pWnd = WindowFromPoint( point );
        if( pWnd == this )
            pWnd = m_pParentWnd;

        hittest = (int)pWnd->SendMessage(WM_NCHITTEST,
                                         0,MAKELONG(point.x,point.y));
        if( hittest == HTCLIENT ) {
            pWnd->ScreenToClient( &point );
            pMsg->lParam = MAKELONG(point.x,point.y);
        } else {
            switch( pMsg->message ) {
            case WM_LBUTTONDOWN:
                pMsg->message = WM_NCLBUTTONDOWN;
                break;
            case WM_RBUTTONDOWN:
                pMsg->message = WM_NCRBUTTONDOWN;
                break;
            case WM_MBUTTONDOWN:
                pMsg->message = WM_NCMBBUTTONDOWN;
                break;
            }
            pMsg->wParam = hittest;
            pMsg->lParam = MAKELONG(point.x,point.y);
        }
        ReleaseCapture();
        ShowWindow( SW_HIDE );
        pWnd->PostMessage( pMsg->message, pMsg->wParam, pMsg->lParam );
        return TRUE;

    case WM_KEYDOWN:
    case WM_SYSKEYDOWN:
        ReleaseCapture();
    }
}

```

```

        ShowWindow( SW_HIDE );
        m_pParentWnd->PostMessage(pMsg->message, pMsg->wParam,
                                   pMsg->lParam );
        return TRUE;
    }

    if( GetFocus() == NULL )
    {
        ReleaseCapture();
        ShowWindow( SW_HIDE );
        return TRUE;
    }

    return CWnd::PreTranslateMessage(pMsg);
}

```

(2) 设计辅助函数 CellRectFromPoint

该函数用于得到鼠标光标下的矩形子项的列索引。该函数的基本思路是首先得到被单击条目的行索引,再在该条目的所有子项中循环以找到单击位置处的列索引。清单 5-63所示为 CellRectFromPoint 函数的源代码,其中 point 为鼠标单击位置,cellrect 将返回子项的边界矩形,col 将返回子项的列索引,该参数可以为 NULL。

清单 5-63 CellRectFromPoint() 函数

```

int CMyListCtrl::CellRectFromPoint(CPoint& point, RECT * cellrect, int * col)
    const
{
    int colnum;

    // 确定列表视图控件为 LVS_REPORT 风格
    if( (GetStyle() & LVS_TYPEMASK) != LVS_REPORT )
        return -1;

    // 得到控件中最顶端和最底端的可见行
    int row = GetTopIndex();
    int bottom = row + GetCountPerPage();
    if( bottom > GetItemCount() )
        bottom = GetItemCount();

    // 得到列数
    CHeaderCtrl * pHeader = (CHeaderCtrl *)GetDlgItem(0);
    int nColumnCount = pHeader->GetItemCount();

    // 在所有可见行中循环
    for( ; row <= bottom; row++ )
    {
        // 得到条目的边界矩形,并检查单击位置是否在其中
        CRect rect;
        GetItemRect( row, &rect, LVIR_BOUNDS );
        if( rect.PtInRect(point) )
        {
            // 现在开始搜寻单击位置处的列

```

```

        for( colnum = 0; colnum < nColumnCount; colnum++ )
        {
            int colwidth = GetColumnWidth(colnum);
            if( point.x >= rect.left &&
                point.x <= (rect.left + colwidth) )
            {
                // 找到单击位置所在的列
                RECT rectClient;
                GetClientRect( &rectClient );
                if( point.x > rectClient.right )
                    return -1;
                if( col )
                    *col = colnum;
                rect.right = rect.left + colwidth;
                if( rect.right > rectClient.right )
                    rect.right = rectClient.right;
                *cellrect = rect;
                return row;
            }
            rect.left += colwidth;
        }
    }
    return -1;
}

```

(3) 在控件(此处为列表视图控件,当然也可以为其他控件)管理类中定制 WM_MOUSEMOVE 消息的响应函数

在 OnMouseMove 函数中使用 CellRectFromPoint 来确定行、列索引以及子项矩形。然后将条目文本信息和矩形传递给内容提示对象,由内容提示对象来决定是否显示内容提示。

标签文本总是与子项边界有一定距离。对于第一列,该距离为条目图像加上一个空格宽度;而对于其他列,该距离则为两个空格字符的宽度。当然,如果列足够宽,那么这个距离就取决于文本调整。下面的代码使用一个硬编码值作为偏移距离,这要比每次都进行计算并将距离保存在一个成员变量中的做法要好一些。清单 5-64 所示为 OnMouseMove 函数的源代码:

清单 5-64 OnMouseMove() 函数

```

void CMyListCtrl::OnMouseMove(UINT nFlags, CPoint point)
{
    if( nFlags == 0 )
    {
        // 允许显示内容提示
        int row, col;
        RECT cellrect;
        row = CellRectFromPoint(point, &cellrect, &col );
        if( row != -1 )
        {

```



```

        // offset = pDC->GetTextExtent(_T(" "), 1).cx * 2;
        int offset = 6;
        if( col == 0 )
        {
            CRect rcLabel;
            GetItemRect( row, &rcLabel, LVIR_LABEL );
            offset = rcLabel.left - cellrect.left + offset / 2;
        }
        cellrect.top--;
        m_titletip.Show( cellrect, GetItemText( row, col ), offset - 1 );
    }
    CListCtrl::OnMouseMove(nFlags, point);
}

```

(4) 在控件管理类中创建内容提示对象

在应用程序所用的控件类,例如 CListView 或 CListCtrl 派生类中加入一个 CTitleTip 成员。如果使用 CListCtrl 派生类,则重载 PreSubclassWindow 函数并添加如清单 5-65 所示的代码。如果使用的是 CListView 派生类,则需要将该函数添加到 OnCreate 或 OnInitialUpdate 函数中。

清单 5-65 PreSubclassWindow() 函数

```

void CMyListCtrl::PreSubclassWindow()
{
    CListCtrl::PreSubclassWindow();

    // Add initialization code
    m_titletip.Create( this );
}

```

5.6.6 改进内容提示

如果不进行列的拖动,那么使用上一节中设计的代码毫无问题,然而在进行了列拖动(改变了列的位置)后上述代码就会出现問題。可以设想,当列的位置发生变化后,由于列只是顺序发生变化而其索引却并不变化,因此内容提示不会因为其位置的列改变而改变其显示内容。在本节中我们将对上一节中设计的代码进行改进,使其也支持列拖动操作。

如果用户进行列拖动,那么应用程序就需要跟踪列的位置。标头控件可以维护当前列位置和原先的列位置。我们可以使用这些信息来在新位置上显示内容提示。出于这一目的,首先在列表视图控件管理类中添加两个新函数:

```

CString GetTrueItemText( int row, int col );
int GetTrueColumnWidth(int nCurrentPosition);

```

注意 如果在非 CListCtrl 或 CListView 派生类中,添加上述函数需要作出一些修改。

清单 5-66 所示为 GetTrueItemText 函数的源代码:

清单 5-66 GetTrueItemText()函数

```

CString CMyListView::GetTrueItemText(int row, int col)
{
    CListCtrl& ctlList = GetListCtrl();

    // 得到标头控件
    CHeaderCtrl* pHeader = (CHeaderCtrl*)ctlList.GetDlgItem(0);
    _ASSERT(pHeader);

    // 得到当前的列数
    int nCount = pHeader->GetItemCount();

    // 得到真正的需要显示其内容提示的列,这将通过与 hi.iOrder 进行对比得到
    for (int x=0; x< nCount; x++)
    {
        HD_ITEM hi = {0};
        hi.mask = HDI_ORDER;

        BOOL bRet = pHeader->GetItem(x,&hi);
        _ASSERT(bRet);
        if (hi.iOrder == col)
        {
            // 得到相应文本
            return ctlList.GetItemText(row,x);
        }
    }

    _ASSERT(FALSE);
    return "我们不会再出错啦!";
}

```

清单 5-67 所示为 GetTrueColumnWidth 函数的源代码:

清单 5-67 GetTrueColumnWidth()函数

```

int CMyListView::GetTrueColumnWidth(int nCurrentPosition)
{
    CListCtrl& ctlList = GetListCtrl();

    CHeaderCtrl* pHeader = (CHeaderCtrl*)ctlList.GetDlgItem(0);
    _ASSERT(pHeader);

    int nCount = pHeader->GetItemCount();

    for (int x=0; x< nCount; x++)
    {
        HD_ITEM hi = {0};
        hi.mask = HDI_WIDTH | HDI_ORDER;

        BOOL bRet = pHeader->GetItem(x,&hi);
        _ASSERT(bRet);
        if (hi.iOrder == nCurrentPosition)
            return hi.cxy;
    }
}

```

```

    }

    _ASSERTE(FALSE);
    return 0;
}

```

接下来需要进行的工作包括：

- 修改 CMyListCtrl::OnMouseMove 函数,将其中调用的 ItemText 替换为 GetTrueItemText 函数。
- 将 CMyListCtrl::OnMouseMove 函数中的下述代码注释掉：

```

if( col == 0 )
{
    CRect rcLabel;
    ctlList.GetItemRect( row, &rcLabel, LVIR_LABEL );
    offset = rcLabel.left - cellrect.left + offset / 2;
}

```

- 将 CellRectFromPoint 函数中的 GetColumnWidth 调用替换为 GetTrueColumnWidth。

5.7 改变列表视图控件的标头显示

列表视图控件的标头一般用于显示列名,在本节中将向读者介绍如何在标头中显示图像。

5.7.1 在标头中显示图像

在默认情况下,标头控件并不使用图像。然而如果在标头中显示图像,将使应用程序看起来更加活泼(如图 5-6 所示)。每个单独的标头条目都可以使用位图,也就是说可以在标头中显示位图,或将位图与标头标签一起显示。调用 CHeaderCtrl::SetItem 函数能够为标头条目指定或移除位图。

下述示范代码为一个标头控件条目指定了一个位图,并且将该位图与标签一起使用。

```

HD_ITEM hditem;
hditem.mask = HDI_FORMAT;
((CHeaderCtrl*)GetDlgItem(0))->GetItem( nCol, &hditem );
hditem.mask = HDI_BITMAP | HDI_FORMAT;
hditem.fmt |= HDF_BITMAP;
// mybitmap 为包含位图的 CBitmap 对象
hditem.hbm = (HBITMAP)mybitmap.GetSafeHandle();
((CHeaderCtrl*)GetDlgItem(0))->SetItem( nCol, &hditem );

```

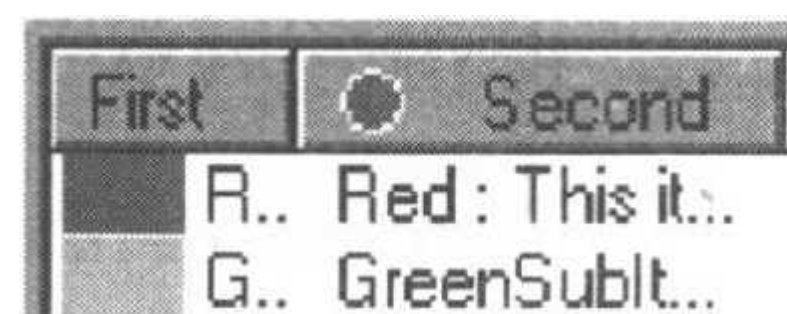


图 5-6 在标头中显示图像

如果希望标头中只显示位图而不显示标签,则应该去除 `fmt` 字段的 `HDF_STRING` 标志,示范代码如下:

```
HD_ITEM hdittem;
hdittem.mask = HDI_FORMAT;
((CHeaderCtrl*)GetDlgItem(0)) -> GetItem( nCol, &hdittem );
hdittem.mask = HDI_BITMAP | HDI_FORMAT;
hdittem.fmt |= HDF_BITMAP;
hdittem.fmt &= ~HDF_STRING;
hdittem.hbm = (HBITMAP)mybitmap.GetSafeHandle();
((CHeaderCtrl*)GetDlgItem(0)) -> SetItem( nCol, &hdittem );
```

要从标头中移除位图,只需要调用 `SetItem` 函数,并清除 `format` 字段中的 `HDF_BITMAP` 标志即可;或者仅仅将 `hbm` 字段设置为 `NULL`,示范代码如下:

```
HD_ITEM hdittem;
hdittem.mask = HDI_FORMAT;
((CHeaderCtrl*)GetDlgItem(0)) -> GetItem( nCol, &hdittem );
hdittem.mask = HDI_FORMAT;
hdittem.fmt &= ~HDF_BITMAP;
((CHeaderCtrl*)GetDlgItem(0)) -> SetItem( nCol, &hdittem );
```

5.7.2 在标头中使用图像列表

虽然能够为标头控件中的每个条目都指定一个位图,然而使用图像列表将更加具有优越性。首先,能够以透明方式(以及其他一些效果)绘制图像;其次,能够使用一个位图管理所有图像,或者可以使用图标。特别是当标头中无法完全显示标头标签时,显示图像将是非常有效的替代方式;此外,不同的图像还可以体现应用程序的不同状态。

由于标头控件并不支持图像列表,因此我们必须手动定制代码。具体实现步骤如下:

- (1) 创建 `CHeaderCtrl` 的派生类 `CMyHeaderCtrl`。
- (2) 添加用于存放图像列表的成员变量 `m_pImageList`,以及某列将显示的图像索引 `m_mapImageIndex`。后一个变量中存放了列号和与其相联系的图像号。

```
protected:
    CImageList* m_pImageList;
    CMap<int, int, int, int> m_mapImageIndex;
```

- (3) 在构造函数中初始变量:将 `m_pImageList` 设置为 `NULL`,这表示尚未设置图像列表。清单 5-68 所示为类的构造函数:

清单 5-68 CMyHeaderCtrl 类的构造函数

```
CMyHeaderCtrl::CMyHeaderCtrl()
{
    m_pImageList = NULL;
}
```

(4) 添加设置和检索图像列表的成员函数。

其中 `SetImageList` 函数简单地更新 `m_pImageList`, 并返回原先的图像列表。而 `GetItemImage` 函数则返回与指定标头条目相关的图像。如果条目没有与其相关的图像, 则返回 `-1`。

`SetItemImage` 函数能够设置图像映射, 并修改标头条目的属性为自绘制。这将确保调用 `DrawItem` 函数, 并且该函数接着会触发标头控件的重新绘制, 以显示改变。清单 5-69、5-70 和 5-71 所示为这三个函数的源代码:

清单 5-69 SetImageList() 函数

```
CImageList * CMyHeaderCtrl::SetImageList( CImageList * pImageList )
{
    CImageList * pPrevList = m_pImageList;
    m_pImageList = pImageList;
    return pPrevList;
}
```

清单 5-70 GetImage() 函数

```
int CMyHeaderCtrl::GetItemImage( int nItem )
{
    int imageIndex;
    if( m_mapImageIndex.Lookup( nItem, imageIndex ) )
        return imageIndex;
    return -1;
}
```

清单 5-71 SetImage() 函数

```
void CMyHeaderCtrl::SetItemImage( int nItem, int nImage )
{
    // 保存图像索引
    m_mapImageIndex[nItem] = nImage;

    // 将条目属性改为自绘制
    HD_ITEM hditem;

    hditem.mask = HDI_FORMAT;
    GetItem( nItem, &hditem );
    hditem.fmt |= HDF_OWNERDRAW;
    SetItem( nItem, &hditem );

    // 触发控件的重绘
    Invalidate();
}
```

(5) 重载 `DrawItem` 函数。

框架会对每个具有 `HDF_OWNERDRAW` 属性的标头条目调用 `DrawItem` 函数。因此我们需要在此函数中添加代码以显示图像列表中的图像。如果绘制了图像, 则标头被移动到图像右侧, 以免覆盖图像。

在 DrawItem 函数中首先挂接作为参数传递给它的设备环境。在函数返回前,必须解除两者之间的连接,这样当 CDC 对象被销毁时设备环境也会被释放。然后保存设备环境,并改变裁剪区域,这样 DrawItem 函数的所有更新操作都将发生在标头条目内。

接着函数计算标头标签需要移动的距离,并且如果条目有与之相联系的图像,则绘制它。此后,在绘制标头标签时应调整其矩形,以免覆盖图像。清单 5-72 所示为 DrawItem 函数的源代码:

清单 5-72 DrawItem() 函数

```
void CMyHeaderCtrl::DrawItem( LPDRAWITEMSTRUCT lpDrawItemStruct )
{
    CDC dc;
    dc.Attach( lpDrawItemStruct -> hDC );
    // 得到列矩形
    CRect rcLabel( lpDrawItemStruct -> rcItem );
    // 保存设备环境
    int nSavedDC = dc.SaveDC();
    // 设置剪切区域,以将绘制操作限制在列中进行
    CRgn rgn;
    rgn.CreateRectRgnIndirect( &rcLabel );
    dc.SelectObject( &rgn );
    rgn.DeleteObject();

    // 移动标签
    int offset = dc.GetTextExtent(_T(" "), 1 ).cx * 2;

    // 绘制图像
    int imageIndex;
    if (m_pImageList &&
        m_mapImageIndex.Lookup( lpDrawItemStruct -> itemID, imageIndex ))
    {
        if( imageIndex != -1 )
        {
            m_pImageList -> Draw(&dc, imageIndex, CPoint( rcLabel.left + offset, offset/3 ), ILD_TRANSPARENT );

            // 调整标签矩形
            IMAGEINFO imageinfo;
            if( m_pImageList -> GetImageInfo( imageIndex, &imageinfo ))
            {
                rcLabel.left += offset/2 +
                    imageinfo.rcImage.right - imageinfo.rcImage.left;
            }
        }
    }

    // 得到文本格式与字体
    TCHAR buf[256];
    HD_ITEM hditem;
```

```

hditem.mask = HDI_TEXT | HDI_FORMAT;
hditem.pszText = buf;
hditem.cchTextMax = 255;

GetItem( lpDrawItemStruct->itemID, &hditem );

// 确定绘制列标签的格式
UINT uFormat = DT_SINGLELINE | DT_NOPREFIX | DT_NOCLIP
               | DT_VCENTER | DT_END_ELLIPSIS ;

if( hditem.fmt & HDF_CENTER )
    uFormat |= DT_CENTER;
else if( hditem.fmt & HDF_RIGHT )
    uFormat |= DT_RIGHT;
else
    uFormat |= DT_LEFT;

// 如果鼠标键被按下,则调整矩形
if( lpDrawItemStruct->itemState == ODS_SELECTED )
{
    rcLabel.left++;
    rcLabel.top += 2;
    rcLabel.right++;
}
rcLabel.left += offset;
rcLabel.right -= offset;

// 绘制列标签
if( rcLabel.left < rcLabel.right )
    dc.DrawText(buf, -1, rcLabel, uFormat);

// 恢复原始设备环境
dc.RestoreDC( nSavedDC );
dc.Detach();
}

```

(6) 在列表视图控件类中添加标头控件成员。

现在已经完成了 CMyHeaderCtrl 类的设计,下面将在 CListCtrl 或 CListView 的派生类中添加标头控件成员,以利用其扩展功能。

```

protected:
    CMyHeaderCtrl m_headerctrl;

```

(7) 将列表视图控件中的标头与 CMyHeaderCtrl 类相联系。

只有在两者间创建了联系之后,才能使列表视图控件中的标头具有 CMyHeaderCtrl 的功能。如果使用 CListView 派生类,则应该在 OnInitialUpdate 函数中进行联系;而如果使用的是 CListCtrl 派生类,则应该在 PreSubclassWindow 函数中进行。无论是那种情况,都必须保证在调用了基类版本的函数后,再执行归类操作。如果列表视图控件不是报表风格,则必须在归类操作前修改控件的风格。清单 5-73 所示为 PreSubClassWindow 函数的源代码:

清单 5-73 PreSubclassWindow() 函数

```
void CMyListCtrl::PreSubclassWindow()
{
    CListCtrl::PreSubclassWindow();

    // Add initialization code
    m_headerctrl.SubclassWindow( ::GetDlgItem(m_hWnd,0) );
}
```

(8) 调用 SetImageList 和 SetItemImage 函数。

在调用 SetItemImage 函数选择列将使用的图像前,应该首先调用 SetImageList 函数以初始化标头控件将使用的图像列表。如果希望移除所有图像,则应该以 NULL 参数调用 SetImageList 函数。如果要从一个标头条目中移除图像,则应该调用 SetItemImage 函数并指定 -1 为图像索引。

本章小结

本章向读者介绍了如何修改常规 Windows 列表视图控件。通过本章学习读者应该达到以下几点:

- 掌握 CListCtrl 类的使用。
- 掌握定制列表视图控件的方法。

第 6 章 树视图控件

树视图控件主要用于显示具有层次结构的条目,例如文件系统等。它也是 Windows 软件中最常用的控件之一,经常与列表视图控件联合使用。

本章要点:

- CTreeCtrl 类的使用;
- 树视图控件的基本操作编程;
- 改善树视图控件的外观;
- 树视图控件内容的序列化。

6.1 树视图控件基础

树视图控件是用于显示具有层次结构条目的窗口,它尤其适于显示目录结构,例如 Windows 操作系统中的“资源管理器”的左半边窗口。树视图控件中的每个条目都由标签和可选的位图组成,而且其中的每个条目都可以拥有一个子条目列表。单击其中的条目,就可以将其下子条目展开或收拢。

CTreeCtrl 类为 Windows 树视图控件提供了功能支持。只有 Windows 95 及 Windows NT 3.51 后的版本为 CTreeCtrl 类提供支持。CTreeCtrl 类的派生结构如图 6-1 所示。

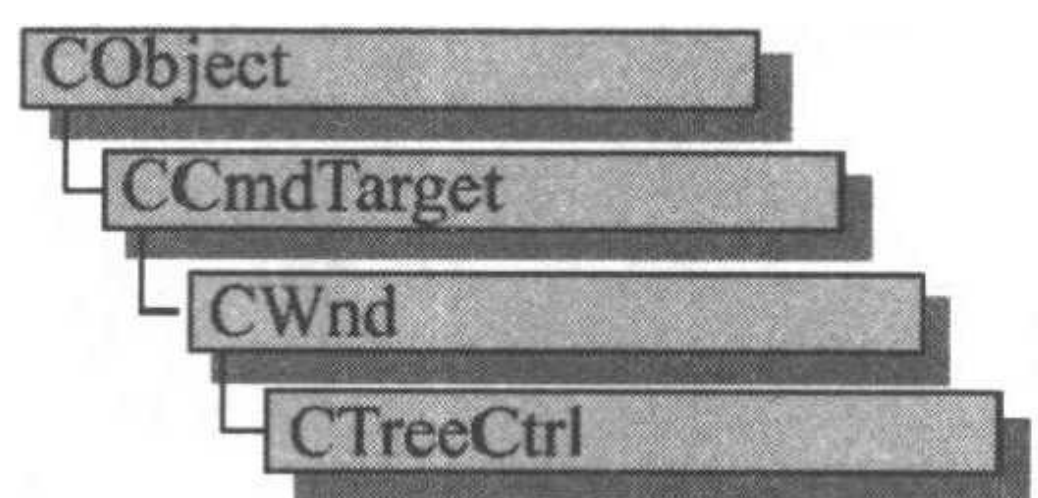


图 6-1 CTreeCtrl 类的派生结构

6.1.1 树视图控件概述

MFC 提供了两个用于封装树视图控件的类:CTreeCtrl 和 CTreeView。这两个类在各自的适用范围内有着各自的优势。当需要使用子窗口控件时使用 CTreeCtrl,例如在对话框应用程序中。而当需要在文档/视图结构的应用程序中以窗口方式使用树视图控件,则应该选择 CTreeView 类。CTreeView 对象将占据框架或分隔窗口的整个客户区。当其父窗口的尺寸改变时,控件的尺寸也会自动变化,而且它能够执行来自菜单、工具栏或加速键的命令。使用 CTreeView 时,对文档并没有特殊要求,也就是说完全可以使用 CDocument 类。

树视图控件典型应用的操作方法如下所示:

- 树视图控件的创建。如果在对话框资源中创建控件,或使用的是 CTreeView,则当对话框或视图创建后,树视图控件会自动被创建。如果希望以子窗口的方式创建控件,则需要调用 Create 函数。

- 如果希望树视图控件使用图像,则调用 SetImageList 为其指定图像列表。此外,也可以调用 SetIndent 改变缩进值。这个工作一般在 OnInitDialog 函数(对话框)或 OnInitialUpdate 函数(视图)中完成。
- 调用 InsertItem 函数能够将条目(数据)插入列表视图控件中。InsertItem 将返回条目的句柄,这样以后可以使用该句柄继续插入子条目。初始化数据的工作一般在 OnInitDialog 函数(对话框)或 OnInitialUpdate 函数(视图)中完成。
- 当用户与控件进行交互时,控件将会发送相应的通告消息。如果应用程序需要处理其中的某个通告,则可以为其在控件窗口消息映射中添加 ON_NOTIFY_REFLECT 宏,或在父窗口消息映射中添加 ON_NOTIFY 宏。树视图控件常用的通告消息如表 6-1 所示:

表 6-1 树视图控件的通告消息

通告消息	含义
TVN_BEGINDRAG	开始拖拽操作
TVN_BEGINLABELEDIT	开始在位编辑条目标签
TVN_BEGINRDRAG	使用鼠标右键开始拖拽操作
TVN_DELETEITEM	删除指定条目
TVN_ENDLABELEDIT	结束编辑条目标签
TVN_GETDISPINFO	请求显示条目所需的信息
TVN_ITEMEXPANDED	父条目被展开或收拢
TVN_ITEMEXPANDING	父条目将被展开或收拢
TVN_KEYDOWN	键盘动作
TVN_SELCHANGED	控件中被选条目发生变化
TVN_SELCHANGING	控件中的被选条目将发生变化
TVN_SETDISPINFO	通告将更新条目数据

- 调用不同的设置函数来设置控件成员值,例如设置和改变条目文本、图像以及数据。
- 调用不同的检索函数来得到控件的内容。
- 当控件使用完毕后,要确定将其销毁。如果在对话框中使用控件(或直接以视图的形式使用),则控件会被自动销毁。否则,需要手动进行销毁。

操作树视图控件的方式取决于控件对象的创建方式。如果在控件在对话框中,则在对话框类中定义 CTreeCtrl 类型的成员,并通过该成员调用控件的成员函数。如果树视图控件为子窗口,则应该使用用以创建控件的 CTreeCtrl 对象。如果使用的 CTreeView 对象,则应该调用 CTreeView::GetTreeCtrl 函数得到对控件对象的引用。

6.1.2 构造函数

CTreeCtrl 类的构造函数包括:CTreeCtrl 和 Create 函数,它们能够完成构造 CTreeCtrl 对象和创建 Windows 树视图控件的操作。

- CTreeCtrl

调用该函数以构造 CTreeCtrl 对象,其原型为:

```
CTreeCtrl();
```

- Create

调用该函数以创建树视图控件并将其与 CTreeCtrl 控件相连接,其原型为:

```
BOOL Create( DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID );
```

返回值:

如果初始化成功,则返回非零值,否则返回零值。

参数:

dwStyle —— 指定了树视图控件的风格,其取值如表 6-2 所示。

表 6-2 dwStyle 参数取值

dwStyle 参数取值	含义
TVS_HASLINES	树视图控件中的子条目与其父条目之间有连线
TVS_LINESATROOT	树视图控件中的子条目与根条目之间有连线
TVS_HASBUTTONS	树视图控件在每个父条目的左边添加一个按钮
TVS_EDITLABELS	树视图控件允许用户编辑其中条目的标签
TVS_SHOWSELALWAYS	当树视图控件失去焦点时,被选条目依然保持选中状态
TVS_DISABLEDROGDROP	树视图控件不能发送 TVN_BEGINDRAG 通告消息
TVS_NOTOOLTIPS	树视图控件没有工具提示
TVS_SINGLEEXPAND	当选中某条目时,该条目会自动被展开;而收拢某条目,也会同时导致该条目被解除选择。如果鼠标使用单击选择条目,并且它处于收拢状态,则条目会被展开;如果条目处于展开状态,则会被收拢

rect —— 指定了树视图控件的尺寸和位置,该参数可以为 CRect 对象或 RECT 结构。

pParentWnd —— 指定了树视图控件的父窗口,通常为对话框。该参数不能为 NULL。

nID —— 指定了树视图控件的 ID。

如果在对话框模板中指定树视图控件,或使用 CTreeView 类,则在视图或对话框创建时树视图控件会自动被创建。如果希望以其他窗口的子窗口的形式创建控件,则可以使用 Create 成员函数,并必须将 WS_VISIBLE 作为控件的风格之一。创建 CTreeCtrl 对象需要两步:首先调用构造函数;然后调用 Create 函数。

6.1.3 属性操作函数

CTreeCtrl 类的属性操作函数包括: GetBkColor、SetBkColor、GetImageList、SetImageList、GetCount、GetItem、SetItem、GetIndent、SetIndent、GetNextItem、GetPrevSiblingItem、GetNextSiblingItem、GetEditCtrl、ItemHasChildren、GetChildItem、GetTextColor、SetTextColor、GetTextBkColor、SetTextBkColor、GetCheck、SetCheck、SetItemState、GetItemState、GetItemText、SetItemText、GetParen-

Item、SetItemData、GetItemData、GetSelectedItem、GetFirstVisibleItem、GetNextVisibleItem、GetPrevVisibleItem、GetDropHighlightItem、GetRootItem、GetItemRect、GetVisibleCount、GetToolTips、SetToolTips、GetInsertMarkColor、SetInsertMarkColor、GetItemHeight、SetItemHeight 和 SetInsertMark, 它们可以完成对树视图控件属性的设置和查询等操作。

- GetCount

调用该函数以得到数视图控件的条目数,其原型为:

```
UINT GetCount( );
```

返回值:

如果函数调用成功,则返回树视图控件的条目数,否则返回 - 1。

- GetIndent

调用该函数以得到控件指定条目与其父条目之间的缩进量(以像素为单位),其原型为:

```
UINT GetIndent( );
```

返回值:

如果函数调用成功,则返回缩进量。

- SetIndent

调用该函数以设置父条目与其子条目之间的缩进量(以像素为单位),其原型为:

```
void SetIndent( UINT nIndent );
```

参数:

nIndent —— 指定了缩进量。如果指定的缩进量比系统定义的最小宽度小,则将该参数设置为系统定义的最小值。

- GetImageList

调用该函数以得到与树视图控件相关的图像列表控件,其原型为:

```
CImageList * GetImageList( UINT nImage );
```

返回值:

如果函数调用成功,则返回控件的图像列表控件指针,否则返回 NULL。

参数:

nImage —— 指定了将得到的图像列表控件类型。如果该参数被设置为 TVSIL_NORMAL,则将得到常规图像列表,其中包括树视图控件条目的选中和未选状态图像。如果参数被设置为 TVSIL_STATE,则将得到状态图像列表,其中包括树视图控件条目的用户定义状态图像。

树视图控件的每个条目都有一对位图图像,其中之一在条目被选中时显示,而另一个则在条目处于未选中状态时显示。例如,在显示目录结构的树视图控件中,某个条目为一个目录,当它被选中时,其图标为打开的文件夹;而在未选中时,其图标为关闭的文件夹。

- SetImageList

调用该函数以设置与树视图控件相关的图像列表控件指针,其原型为:

```
CImageList * SetImageList( CImageList * pImageList, int nImageListType );
```

返回值:

如果函数调用成功,则返回先前控件的图像列表控件指针,否则返回 NULL。

参数:

pImageList —— 指定了将设置的图像列表控件指针。如果该参数为 NULL,则将树视图控件的所有图像去除。

nImageListType —— 指定了图像列表控件的类型,其取值可以为:TVSIL_NORMAL 和 TVSIL_STATE。

• GetNextItem

调用该函数以得到符合指定条件的下一个树视图控件条目,其原型为:

```
HTREEITEM GetNextItem( HTREEITEM hItem, UINT nCode );
```

返回值:

如果函数调用成功,则返回下一个条目的句柄,否则返回 NULL。

参数:

hItem —— 指定了用于参照的树视图条目句柄。

nCode —— 指定了检索操作与 hItem 的关系,其取值如表 6-3 所示:

表 6-3 nCode 参数取值

nCode 参数取值	含义
TVGN_CARET	获得当前被选条目
TVGN_CHILD	获得第一个子条目,此时 hItem 必须为 NULL
TVGN_DROPHILITE	获得拖放操作(drag-and-drop operation)的目的条目
TVGN_FIRSTVISIBLE	获得第一个可见条目
TVGN_NEXT	获得下一个同级条目
TVGN_NEXTVISIBLE	获得指定项目的下一个可见条目
TVGN_PARENT	获得指定条目的父条目
TVGN_PREVIOUS	获得前一个同级条目
TVGN_PREVIOUSVISIBLE	获得指定项目的前一个可见条目
TVGN_ROOT	获得指定条目的根条目的第一个子条目

• ItemHasChildren

调用该函数以确定指定条目是否具有子条目,其原型为:

```
BOOL ItemHasChildren( HTREEITEM hItem );
```

返回值:

如果指定条目具有子条目,则返回非零值,否则返回零值。

参数:

hItem —— 指定了条目句柄。

• GetChildItem

调用该函数以得到指定条目的子条目,其原型为:

```
HTREEITEM GetChildItem( HTREEITEM hItem );
```

返回值:

如果函数调用成功,则返回子条目句柄,否则返回 NULL。

参数:

hItem —— 指定了条目句柄。

- **GetNextSiblingItem**

调用该函数以得到指定树视图条目的下一个同级条目,其原型为:

```
HTREEITEM GetNextSiblingItem( HTREEITEM hItem );
```

返回值:

如果函数调用成功,则返回下一个同级条目句柄,否则返回 NULL。

参数:

hItem —— 指定了条目句柄。

- **GetPrevSiblingItem**

调用该函数以得到指定树视图条目的上一个同级条目,其原型为:

```
HTREEITEM GetPrevSiblingItem( HTREEITEM hItem );
```

返回值:

如果函数调用成功,则返回上一个同级条目句柄,否则返回 NULL。

参数:

hItem —— 指定了条目句柄。

- **GetParentItem**

调用该函数以得到指定树视图条目的父条目,其原型为:

```
HTREEITEM GetParentItem( HTREEITEM hItem );
```

返回值:

如果函数调用成功,则返回指定条目的父条目句柄,否则返回 NULL。

参数:

hItem —— 指定了条目句柄。

- **GetFirstVisibleItem**

调用该函数以得到指定树视图的第一个可见条目,其原型为:

```
HTREEITEM GetFirstVisibleItem( );
```

返回值:

如果函数调用成功,则返回第一个可见条目的句柄,否则返回 NULL。

- **GetNextVisibleItem**

调用该函数以得到指定树视图条目的下一个可见条目,其原型为:

```
HTREEITEM GetNextVisibleItem( HTREEITEM hItem );
```

返回值:

如果函数调用成功,则返回下一个可见条目的句柄,否则返回 NULL。

参数:

hItem —— 指定了条目句柄。

- GetPrevVisibleItem

调用该函数以得到指定树视图条目的上一个可见条目,其原型为:

```
HTREEITEM GetPrevVisibleItem( HTREEITEM hItem );
```

返回值:

如果函数调用成功,则返回上一个可见条目的句柄,否则返回 NULL。

参数:

hItem —— 指定了条目句柄。

- GetSelectedItem

调用该函数以得到当前被选中的树视图控件条目,其原型为:

```
HTREEITEM GetSelectedItem( );
```

返回值:

如果函数调用成功,则返回被选条目的句柄,否则返回 NULL。

- GetDropHighlightItem

调用该函数以得到拖放操作的目标,其原型为:

```
HTREEITEM GetDropHighlightItem( );
```

返回值:

如果函数调用成功,则返回目标条目的句柄,否则返回 NULL。

- GetRootItem

调用该函数以得到指定树视图条目的根条目,其原型为:

```
HTREEITEM GetRootItem( );
```

返回值:

如果函数调用成功,则返回根条目句柄,否则返回 NULL。

- GetItem

调用该函数以得到指定树视图条目的属性,其原型为:

```
BOOL GetItem( TVITEM* pItem );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

pItem —— 为指向 TVITEM 结构的句柄。它用于指定或接收树视图控件条目的属性,

它与 TV_ITEM 结构等价,只是名称不同,新应用程序都应该使用新名称。TVITEM 结构的定义如下:

```
typedef struct tagTVITEM{
    UINT mask;
    HTREEITEM hItem;
    UINT state;
    UINT stateMask;
    LPTSTR pszText;
    int cchTextMax;
    int iImage;
    int iSelectedImage;
    int cChildren;
    LPARAM lParam;
} TVITEM, FAR * LPTVITEM;
```

结构成员:

mask —— 指定了结构的哪些成员有效。当在 TVM_GETITME 消息中使用该结构时,mask 成员用于表示将获取条目的哪些属性。mask 成员的取值如表 6-4 所示。

表 6-4 mask 成员取值

mask 成员取值	含义
TVIF_CHILDREN	cChildren 成员有效
TVIF_HANDLE	hItem 成员有效
TVIF_IMAGE	iImage 成员有效
TVIF_PARAM	lParam 成员有效
TVIF_SELECTEDIMAGE	iSelectedImage 成员有效
TVIF_STATE	state 和 stateMask 成员有效
TVIF_TEXT	pszText 和 cchTextMax 成员有效

hItem —— 指定了结构信息的来源条目。

state —— 指定了条目状态,由一套位标志和图像列表索引组成。当设置一个条目时,stateMask 成员表示其有效位。当接收条目状态时,该成员将返回由 stateMask 成员指定的状态位。

成员的 0 到 7 位包含了条目状态标志,条目的状态标志如表 6-5 所示。

表 6-5 条目状态标志

条目状态标志	含义
TVIS_BOLD	条目被加粗
TVIS_CUT	条目作为剪切 - 粘贴操作的一部分被选中
TVIS_DROPHILITED	条目作为拖动操作的目标被选中

续表

条目状态标志	含义
TVIS_EXPANDED	条目的子条目列表当前处于打开状态,也就是说子条目可见。该值只能用于父条目
TVIS_EXPANDEDONCE	条目的子条目至少被打开过一次。设置了该标志的父条目在响应 TVM_EXPAND 消息时,不会产生 TVN_ITEMEXPANDING 和 TVN_ITEMEXPANDED 通告消息。与 TVM_EXPAND 一起使用 TVE_COLLAPSE 和 TVE_COLLAPSERESET 将导致本状态被重置。该值只能用于父条目
TVIS_EXPANDPARTIAL	条目被部分打开。在这种状态下,子条目中只有一部分是可见的,此时父条目的加号符也同时显示
TVIS_SELECTED	条目被选中。此时条目的外观与其是否具有焦点有关,而且将使用系统颜色绘制条目

当设置或接收条目的覆盖图像索引或状态图像索引时,必须在 stateMask 成员中指定表 6-6 中所示标志。这些值也可以用于关闭不用的状态位。

表 6-6 条目状态标志

条目状态标志	含义
TVIS_OVERLAYMASK	用于指定条目覆盖图像索引的位掩码
TVIS_STATEIMAGEMASK	用于指定条目状态图像索引的位掩码
TVIS_USERMASK	与 TVIS_STATEIMAGEMASK 一致

成员的 8 到 11 位指定了覆盖图像索引。覆盖图像用于显示在条目图标图像之上。如果这些位为 0,则表示条目没有覆盖图像。使用 TVIS_OVERLAYMASK 掩码可以分离出这些位。要设置该成员的覆盖图像索引,应该使用 INDEXTOOVERLAYMASK 宏。而图像列表的覆盖图像,则由 ImageList::SetOverlayImage 函数设置。

成员的 12 到 15 指定了状态图像掩码。状态图像显示在条目图标的旁边,以标识应用程序定义的状态。如果这些位为 0,则条目不具有状态图像。使用 TVIS_STATEIMAGEMASK 掩码可以分离出这些位。使用 INDEXTOSTATEIMAGEMASK 宏,能够设置状态图像索引。状态图像索引指定了状态图像列表中将绘制的图像索引。状态图像列表由 TVM_SETIMAGELIST 消息指定。

stateMask —— 指定了 state 成员的有效位。在接收条目状态时,将 stateMask 成员的位设置为将返回的位。而在设置条目状态时,将 stateMask 成员的位设置为 state 成员中将改变的位。例如,要设置或接收条目的覆盖图像索引,则应该设置 TVIS_OVERLAYMASK 位;要设置或接收条目状态图像索引,则应该设置 TVIS_STATEIMAGEMASK 位。

pszText —— 指定了条目文本。如果该成员为 LPSTR_TEXTCALLBACK,则父窗口将负责储存其名称。在这种情况下,当需要显示、排序或编辑条目文本时,树视图控件向父窗口发送 TVN_GETDISPINFO 通告消息;当需要设置条目文本时,则向父窗口发送 TVN_SETDISPINFO 通告消息。

当结构用于接收条目属性时,该成员指定了将返回条目文本的缓冲区地址。

cchTextMax —— 指定了 pszText 成员的长度(以字符为单位)。如果结构是用于设置条目文本的,则该成员被忽略。

iImage —— 指定了当条目处于未选中状态时将使用的树视图控件图标图像列表的索引。

如果该成员为 L_IMAGECALLBACK,则父窗口负责储存索引。在这种情况下,当需要显示图像时,树视图控件向父窗口发送 TVN_GETDISPINFO 通告消息。

iSelectedImage —— 指定了当条目处于选中状态时将使用的树视图控件图标图像列表的索引。

如果该成员为 L_IMAGECALLBACK,则父窗口负责储存索引。在这种情况下,当需要显示图像时,树视图控件向父窗口发送 TVN_GETDISPINFO 通告消息。

cChildren —— 指定了条目是否有相关的子条目,该成员的取值如表 6-7 所示:

表 6-7 cChildren 成员取值

cChildren 成员取值	含义
0	条目没有子条目
1	条目有子条目
L_CHILDRENCALLBACK	父窗口负责监控条目是否有子条目。此时,当树视图控件需要显示条目时,就会向父窗口发送 TVN_GETDISPINFO 通告消息,并确定条目是否具有子条目。如果树视图控件具有 TVS_HASBUTTONS 风格,则它使用该成员来确定是否显示按钮,以标识是否存在子条目。用户可以使用该成员以强制控件显示按钮,即使条目没有任何子条目。这允许用户在条目可见或打开时,插入子条目以最小化内存使用的情况下显示按钮

lParam —— 指定了与条目有关的 32 位值。

• SetItem

调用该函数以设置指定树视图条目的属性,其原型为:

```
BOOL SetItem( TVITEM* pItem );  
BOOL SetItem( HTREEITEM hItem, UINT nMask, LPCTSTR lpszItem, int nImage, int nSelectedImage,  
UINT nState, UINT nStateMask, LPARAM lParam );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

pItem —— 指定了包含新条目属性的 TVITEM 结构。

hItem —— 指定了将设置其属性的条目句柄。

nMask —— 指定了将设置的属性。

lpszItem —— 指定了条目文本。

nImage —— 指定了条目图像索引。

nSelectedImage —— 指定了条目被选中时显示的图像索引。

nState —— 指定了条目的状态值。

nStateMask —— 指定了将设置的状态。

lParam —— 指定了与条目有关的 32 位应用程序定义值。

本函数的参数实际与 TVITEM 结构的成员具有对应关系,请读者参照 GetItem 函数中对该结构的介绍并加以对比。

- GetItemState

调用该函数以得到条目状态,其原型为:

```
UINT GetItemState( HTREEITEM hItem, UINT nStateMask ) const;
```

返回值:

如果调用成功,则返回条目的状态。

参数:

hItem —— 指定了将获取其状态的条目句柄。

nStateMask —— 指定了将获取那些状态。

- SetItemState

调用该函数以设置条目句柄,其原型为:

```
BOOL SetItemState( HTREEITEM hItem, UINT nState, UINT nStateMask );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

hItem —— 指定了将设置其状态的条目句柄。

nState —— 指定了条目的新状态。

nStateMask —— 指定了那些条目状态将被设置。

- GetItemImage

调用该函数以得到与条目相关的图像,其原型为:

```
BOOL GetItemImage( HTREEITEM hItem, int& nImage, int& nSelectedImage ) const;
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

hItem —— 指定了将获取其图像的条目句柄。

nImage —— 将返回条目图像索引。

nSelectedImage —— 将返回条目的选中图像索引。

- SetItemImage

调用该函数以设置指定条目的图像,其原型为:

```
BOOL SetItemImage( HTREEITEM hItem, int nImage, int nSelectedImage );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

hItem —— 指定了将设置其图像的条目句柄。

nImage —— 指定了条目图像索引。

nSelectedImage —— 指定了条目选中图像索引。

- GetItemText

调用该函数以得到指定条目的文本,其原型为:

```
CString GetItemText( HTREEITEM hItem ) const;
```

返回值:

如果函数调用成功,则返回包含条目文本的 CString 对象。

参数:

hItem —— 指定了将获取其文本的条目句柄。

- SetItemText

调用该函数设置条目文本,其原型为:

```
BOOL SetItemText( HTREEITEM hItem, LPCTSTR lpszItem );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

hItem —— 指定了将设置其文本的条目句柄。

lpszItem —— 指定了将设置的文本。

- GetItemData

调用该函数以得到与条目相关的、由应用程序定义的 32 位值,其原型为:

```
DWORD GetItemData( HTREEITEM hItem ) const;
```

返回值:

如果函数调用成功,则返回与条目相关的 32 位应用程序定义值。

参数:

hItem —— 指定了将获取其数据的条目句柄。

- SetItemData

调用该函数以设置与条目相关的、由应用程序定义的 32 位值,其原型为:

```
BOOL SetItemData( HTREEITEM hItem, DWORD dwData );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

hItem —— 指定了将设置其数据的条目句柄。

dwData —— 指定了将设置的数据。

- **GetItemRect**

调用该函数以得到指定树视图控件条目的边界矩形,其原型为:

```
BOOL GetItemRect( HTREEITEM hItem, LPRECT lpRect, BOOL bTextOnly );
```

返回值:

如果条目可见,则返回非零值,否则返回零值。

参数:

hItem —— 指定了将获取其边界矩形的条目句柄。

lpRect —— 将返回条目的边界矩形。该矩形的坐标是相对于控件的左上角坐标的。

bTextOnly —— 如果该参数为非零值,则返回条目文本的边界矩形,否则返回条目所占据的整行的边界矩形。

- **GetEditControl**

调用该函数以得到用于编辑指定树视图控件条目的编辑控件句柄,其原型为:

```
CEdit * GetEditControl( );
```

返回值:

如果函数调用成功,则返回用于编辑条目文本的编辑控件的句柄,否则返回 NULL。

- **GetVisibleCount**

调用该函数以得到树视图控件中的可见条目数,其原型为:

```
UINT GetVisibleCount( );
```

返回值:

如果函数调用成功,则返回控件中的可见条目数,否则返回 -1。

- **GetToolTips**

调用该函数以得到树视图控件所用的工具提示控件句柄,其原型为:

```
CToolTipCtrl * GetToolTips( );
```

返回值:

如果函数调用成功,则返回树视图控件所用的 CToolTipCtrl 对象句柄。如果控件具有 TVS_NOTOOLTIPS 风格,则返回 NULL。

- **SetToolTips**

调用该函数以设置树视图控件所用的工具提示控件,其原型为:

```
void SetToolTips( CToolTipCtrl * pWndTip );  
CToolTipCtrl * SetToolTips( CToolTipCtrl * pWndTip );
```

返回值:

如果函数调用成功,则返回树视图控件先前所用的 CToolTipCtrl 对象句柄。如果控件先前没有使用工具提示,则返回 NULL。

参数:

pWndTip —— 指定了树视图控件将使用的 CToolTipCtrl 对象。

- GetBkColor

调用该函数以得到控件当前所用的背景颜色,其原型为:

```
COLORREF GetBkColor( ) const;
```

返回值:

如果函数调用成功,则返回控件的背景色。如果返回值为 -1,则表示控件使用系统颜色作为背景色。

- SetBkColor

调用该函数以设置控件的背景色,其原型为:

```
COLORREF SetBkColor( COLORREF clr );
```

返回值:

如果函数调用成功,则返回控件先前使用的背景色。如果返回值为 -1,则表示控件使用系统颜色作为背景色

参数:

clr —— 指定了控件的新背景色。

- GetItemHeight

调用该函数以得到当前树视图控件条目的高度,其原型为:

```
SHORT GetItemHeight( ) const;
```

返回值:

如果函数调用成功,则返回条目的高度。

- SetItemHeight

调用该函数以设置树视图控件条目的高度,其原型为:

```
SHORT SetItemHeight( SHORT cyHeight );
```

返回值:

如果函数调用成功,则返回条目的先前高度。

参数:

cyHeight —— 指定了条目的新高度(以像素为单位)。如果该参数小于图像的高度,则它将被设置为图像的高度。如果该参数非偶数,则将被减少到最近的偶数。如果该参数为 -1,则使用默认高度。

- GetTextColor

调用该函数以得到控件的当前文本颜色,其原型为:

```
COLORREF GetTextColor( ) const;
```

返回值:

如果函数调用成功,则返回控件当前所用的文本颜色。如果返回 -1,则控件使用系统颜色作为文本颜色。

- **SetTextColor**

调用该函数以设置控件的文本颜色,其原型为:

```
COLORREF SetTextColor( COLORREF clr );
```

返回值:

如果函数调用成功,则返回先前控件所用的文本颜色。如果返回 -1,则控件使用系统颜色作为文本颜色。

参数:

clr —— 指定了控件的新文本颜色。如果该参数为 -1,则控件使用系统颜色作为文本颜色。

- **SetInsertMark**

调用该函数以在树视图控件中设置插入标记,其原型为:

```
BOOL SetInsertMark( HTREEITEM hItem, BOOL fAfter = TRUE );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

hItem —— 指定了将设置插入标记的条目句柄。如果该参数为 NULL,则将去除插入标记。

fAfter —— 指定了插入标记将置于指定条目之前还是之后。如果该参数为非零值,则插入标记将被置于条目之后,否则将置于条目之前。

- **GetCheck**

调用该函数以得到树视图控件指定条目的复选状态,其原型为:

```
BOOL GetCheck( HTREEITEM hItem ) const;
```

返回值:

如果指定条目被复选,则返回非零值,否则返回零值。

参数:

hItem —— 指定了将接收其状态信息的条目句柄。

- **SetCheck**

调用该函数以设置树视图控件指定条目的复选状态,其原型为:

```
BOOL SetCheck( HTREEITEM hItem, BOOL fCheck = TRUE );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

hItem —— 指定了将改变其复选状态的条目句柄。

fCheck —— 指定了条目应该被设置为复选还是非复选状态。默认情况下,函数将条目设置为复选状态。

- GetInsertMarkColor

调用该函数以得到用于绘制插入标记的颜色,其原型为:

```
COLORREF GetInsertMarkColor( ) const;
```

返回值:

如果函数调用成功,则返回当前插入标记的颜色。

- SetInsertMarkColor

调用该函数以设置用于绘制插入标记的颜色,其原型为:

```
COLORREF SetInsertMarkColor( COLORREF clrNew );
```

返回值:

如果函数调用成功,则返回先前插入标记的颜色。

参数:

clrNew —— 指定了将用于绘制插入标记的新颜色。

6.1.4 常规操作函数

CTreeCtrl 类的常规操作函数包括: InsertItem、DeleteItem、DeleteAllItems、Expand、Select、SelectItem、SelectDropTarget、SelectSetFirstVisible、SortChildren、SortChildrenCB、HitTest、EnsureVisible、EditLabel 和 CreateDragImage,它们可以完成向树视图控件中插入新条目、展开树等操作。

- InsertItem

调用该函数以向树视图控件插入新条目,其原型为:

```
HTREEITEM InsertItem( LPTVINSERTSTRUCT lpInsertStruct );  
HTREEITEM InsertItem( UINT nMask, LPCTSTR lpszItem, int nImage, int nSelectedImage, UINT  
nState, UINT nStateMask, LPARAM lParam, HTREEITEM hParent, HTREEITEM hInsertAfter );  
HTREEITEM InsertItem( LPCTSTR lpszItem, HTREEITEM hParent = TVI_ROOT, HTREEITEM  
hInsertAfter = TVI_LAST );  
HTREEITEM InsertItem( LPCTSTR lpszItem, int nImage, int nSelectedImage,  
HTREEITEM hParent = TVI_ROOT, HTREEITEM hInsertAfter = TVI_LAST );
```

返回值:

如果函数调用成功,则返回新条目的句柄,否则返回 NULL。

参数:

lpInsertStruct —— 指定了包含将插入的条目属性的 TVINSERTSTRUCT 结构。该结构与 TVM_INSERTITEM 消息一起使用,它与 TV_INSERTSTRUCT 结构等价,但是新应用程序应该使用当前命名。

nMask —— 指定了将设置的条目属性,请参见在 TVITEM 结构中的介绍。

- lpszItem —— 指定了条目文本。
- nImage —— 指定了条目图像索引。
- nSelectedImage —— 指定了条目选中图像索引。
- nState —— 指定了条目的状态。
- nStateMask —— 指定了将设置的状态,请参见在 TVITEM 结构中的介绍。
- lParam —— 指定了与条目相关的、应用程序指定的 32 位值。
- hParent —— 指定了插入条目的父条目句柄。
- hInsertAfter —— 指定了新条目将插入于其后的条目句柄。

TVINSERTSTRUCT 结构的定义如下:

```
typedef struct tagTVINSERTSTRUCT {
    HTREEITEM hParent;
    HTREEITEM hInsertAfter;
    # if (_WIN32_IE >= 0x0400)
        union
        {
            TVITEMEX itemex;
            TVITEM item;
        } DUMMYUNIONNAME;
    # else
        TVITEM item;
    # endif
} TVINSERTSTRUCT, FAR * LPTVINSERTSTRUCT;
```

结构成员:

hParent —— 指定了父条目句柄。如果该参数为 TVL_ROOT 或 NULL,则该条目将被插入于控件的根部。

hInsertAfter —— 指定了新条目将插入于其后的条目句柄,或该参数也可取表 6-8 中的值。

表 6-8 hInsertAfter 成员取值

hInsertAfter 成员取值	含义
TVL_FIRST	插入条目处于列表起始处
TVL_LAST	插入条目处于列表末尾
TVL_SORT	插入条目将以字母顺序插入列表中

itemex —— 指定了包含新条目信息的 TVITEMEX 结构。该结构只比 TVITEM 结构多出一个成员 iIntegral,该成员用于表示条目的高度。

item —— 指定了包含新条目信息的 TVITEM 结构。

- DeleteItem

调用该函数以删除树视图控件中的指定条目,其原型为:

```
BOOL DeleteItem( HTREEITEM hItem );
```


返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

hItem —— 指定了将被删除的条目句柄。如果该参数为 TVL_ROOT,则所有条目都将被删除。

- DeleteAllItems

调用该函数以删除树视图控件中的所有条目,其原型为:

```
BOOL DeleteAllItems( );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

- Expand

调用该函数以展开或收拢指定条目的子条目,其原型为:

```
BOOL Expand( HTREEITEM hItem, UINT nCode );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

hItem —— 指定了将打开或收拢的条目句柄。

nCode —— 指定了采取的操作,其取值如表 6-9 所示。

表 6-9 hInsertAfter 成员取值

hInsertAfter 成员取值	含义
TVE_COLLAPSE	收拢列表
TVE_COLLAPSERESET	收拢列表,并删除子条目
TVE_EXPAND	打开列表
TVE_TOGGLE	如果当前列表已经打开,则收拢之;如果当前列表已经收拢,则打开之

- Select

调用该函数以选中、滚动视图,或重新绘制指定条目,其原型为:

```
BOOL Select( HTREEITEM hItem, UINT nCode );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

hItem —— 指定了条目句柄。

nCode —— 指定了操作类型,其取值如表 6-10 所示。

表 6-10 hInsertAfter 成员取值

hInsertAfter 成员取值	含义
TVGN_CARET	选中指定条目
TVGN_DROPHILITE	以拖放操作的目标风格重新绘制指定条目
TVGN_FIRSTVISIBLE	垂直滚动控件,以使指定条目控件可见

如果 nCode 参数为 TVGN_CARET,则父窗口将收到 TVN_SELCHANGING 和 TVN_SELCHANGED 通告消息。而且,如果指定条目为收拢的父条目下的子条目,则该父条目将被展开以使指定条目可见。在这种情况下,父窗口将收到 TVN_ITEMEXPANDING 和 TVN_ITEMEXPANDED 通告消息。

• SelectItem

调用该函数以选中指定条目,其原型为:

```
BOOL SelectItem( HTREEITEM hItem );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

hItem —— 指定了条目句柄。

• SelectDropTarget

调用该函数以拖放操作的目标方式重新绘制指定条目,其原型为:

```
BOOL SelectDropTarget( HTREEITEM hItem );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

hItem —— 指定了条目句柄。

• SelectSetFirstVisible

调用该函数以将指定条目作为第一个可见条目加以选中,其原型为:

```
BOOL SelectSetFirstVisible( HTREEITEM hItem );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

hItem —— 指定了将被设置为第一个可见条目的句柄。

SelectSetFirstVisible 函数将向父窗口发送 TVM_SELECTITEM 和 TVGN_FIRSTVISIBLE 消息。

• EditLabel

调用该函数以在位编辑指定树视图控件条目,其原型为:

```
CEdit * EditLabel( HTREEITEM hItem );
```

返回值：
如果函数调用成功，则返回用于编辑条目文本的 CEdit 对象，否则返回 NULL。
参数：

hItem —— 指定了将被编辑的条目句柄。
• HitTest
调用该函数以得到鼠标当前位置处的条目句柄，其原型为：

```
HTREEITEM HitTest( CPoint pt, UINT * pFlags );  
HTREEITEM HitTest( TVHITTESTINFO * pHitTestInfo );
```

返回值：
如果函数调用成功，则返回指定位置处的条目句柄。如果指定位置处没有条目存在，则返回 NULL。
参数：
pt —— 指定将用于测试的客户区点坐标。
pFlags —— 指定了将用于接收测试结果的变量，其取值如表 6-11 所示。

表 6-11 pFlags 参数取值

pFlags 参数取值	含义
TVHT_ABOVE	指定点在客户区上
TVHT_BELOW	指定点在客户区下
TVHT_NOWHERE	指定点在客户区中，但在最后一个条目之下
TVHT_ONITEM	指定点在与条目相关的位图或标签上
TVHT_ONITEMBUTTON	指定点在与条目相关的按钮上
TVHT_ONITEMICON	指定点在与条目相关的位图上
TVHT_ONITEMINDENT	指定点在与条目相关的缩进上
TVHT_ONITEMLABEL	指定点在与条目相关的标签(字符串)上
TVHT_ONITEMRIGHT	指定点在与条目右边
TVHT_ONITEMSTATEICON	指定点在与条目相关的状态图标上
TVHT_TOLEFT	指定点在客户区的左边
TVHT_TORIGHT	指定点在客户区的右边

pHitTestInfo —— 指定了包含单击测试信息、并返回测试结果的 TVHITTESTINFO 结构。该结构与 TVM_HITTEST 消息共同使用，它与 TV_HITTESTINFO 结构等价，不过我们应该遵守新的命名。TVHITTESTINFO 结构的定义如下：

```
typedef struct tagTVHITTESTINFO {  
    POINT pt;  
    UINT flags;  
    HTREEITEM hItem;  
} TVHITTESTINFO, FAR * LPTVHITTESTINFO;
```

结构成员：

pt —— 指定了用于测试的点坐标。

flags —— 将返回测试结果,其取值参见表。

hItem —— 将返回指定位置处的条目句柄。

- **CreateDragImage**

调用该函数用于为指定条目创建拖动位图,其原型为：

```
CImageList * CreateDragImage( HTREEITEM hItem );
```

返回值：

如果函数调用成功,则返回将添加拖动图像的图像列表指针,否则返回 NULL。

参数：

hItem —— 指定了将被拖动的条目句柄。

- **SortChildren**

调用该函数以排序指定父条目下的子条目,其原型为：

```
BOOL SortChildren( HTREEITEM hItem );
```

返回值：

如果函数调用成功,则返回非零值,否则返回零值。

参数：

hItem —— 指定了将排序其子条目的父条目句柄。如果该参数为 NULL,则将从控件根部开始排序。

- **EnsureVisible**

调用该函数以使指定条目可见,其原型为：

```
BOOL EnsureVisible( HTREEITEM hItem );
```

返回值：

如果系统滚动控件以使条目控件可见,则返回 TRUE,否则返回 FALSE。

参数：

hItem —— 指定了将使其可见的条目句柄。

- **SortChildrenCB**

调用该函数以使用应用程序定义的函数,排序指定条目的子条目,其原型为：

```
BOOL SortChildrenCB( LPTVSORTCB pSort );
```

返回值：

如果函数调用成功,则返回非零值,否则返回零值。

参数：

pSort —— 为指向 TVSORTCB 结构的指针。该结构与 TVM_SORTCHILDRENCB 消息共同使用,它与 TV_SORTCB 结构等价,不过应该使用新命名。TVSORTCB 结构的定义如下：

```
typedef struct tagTVSORTCB{
```

```
HTREEITEM hParent;  
PFNTVCOMPARE lpfnCompare;  
LPARAM lParam;  
} TVSORTCB, FAR * LPTVSORTCB;
```

结构成员：

hParent —— 指定了父条目的句柄。

lpfnCompare —— 指定了应用程序定义的排序函数。在排序中的每一次比较操作中，都会调用该回调函数，该函数应该有以下形式：

```
int CALLBACK CompareFunc(LPARAM lParam1, LPARAM lParam2, LPARAM lParamSort);
```

如果第一个条目应该排在第二个条目之前，则函数返回负值，否则函数应该返回正值。函数的 lParam1 和 lParam2 参数与 TVITEM 结构中的 lParam 成员相关，函数就是使用它们进行比较的。而 lParamSort 参数则与本结构中的 lParam 成员相关。

lParam —— 指定了应用程序定义的 32 位值。

6.2 条目基本操作编程

树视图控件中的每个条目都有自己的子条目列表。拥有一个或多个子条目的条目被称为父条目。子条目在其父条目下显示，并会有一定的缩进量以标识其从属地位。

父条目下的子条目列表可以在任何时候被展开或收拢。当父条目处于展开状态时，从属于它的子条目将在其下显示。而当父条目处于收拢状态时，其子条目将不会被显示。当双击父条目时，或当控件具有 TVS_HASBUTTONS 风格，而用户单击与父条目相关的按钮时，其子条目列表会自动在展开和收拢态间切换。应用程序可以通过调用 Expand 函数来展开或收拢某条目。

调用 InsertItem 函数可以将条目插入控件中。该函数将返回 HTREEITEM 值，该值唯一地标识了被插入的条目。当在添加条目时，必须指定新条目的父条目。如果指定的是 NULL 或 TVL_ROOT，则新添加的条目就被设置为根条目。

当父条目被展开或收拢时，控件会发送 TVN_ITEMEXPANDING 通告消息。使用该通告消息，应用程序能够阻止相应操作或设置父条目与子条目相关属性。在条目被展开或收拢后，控件会发送 TVN_ITEMEXPANDED 通告消息。

当子条目被展开时，它们相对于其父条目有一定的缩进量。调用 SetIndent 可以设置缩进量，而调用 GetIndent 则能够检索当前缩进量。

6.2.1 展开分支

CTreeCtrl 类本身具有能够一次展开一级纲要的成员函数。不过，如果希望完全展开一个分支，那么就on须定制自己的代码。清单 6-1 所示为 ExpandBranch 函数的源代码，其中 hti 为将展开的树视图条目句柄：

清单 6-1 ExpandBranch() 函数

```
void CTreeCtrlEX::ExpandBranch( HTREEITEM hti )
{
    if( ItemHasChildren( hti ) ){
        Expand( hti, TVE_EXPAND );
        hti = GetChildItem( hti );
        do{
            ExpandBranch( hti );
        }while( (hti = GetNextSiblingItem( hti )) != NULL );
    }
    EnsureVisible( GetSelectedItem() );
}
```

在函数中通过递归,将分支下的所有条目都展开。而在函数的最后调用 `EnsureVisible` 函数,以确保被选择的条目是可见的。因为将某个分支下的条目全部打开后,可能会使原来可见的条目不再可见。

6.2.2 收拢分支

使用过“资源管理器”的读者可能会注意到:当将某个条目收拢又再次展开时,其展开状态依然保持为上次的展开状态。虽然这个特性非常好,但是有时用户希望能够将某个分支下的所有条目都收拢,以便能在查看细节前看到更多的内容。清单 6-2 所示为完成此功能的 `CollapseBranch` 函数的源代码:

清单 6-2 CollapseBranch() 函数

```
void CTreeCtrlEX::CollapseBranch( HTREEITEM hti )
{
    if( ItemHasChildren( hti ) ){
        Expand( hti, TVE_COLLAPSE );
        hti = GetChildItem( hti );
        do{
            CollapseBranch( hti );
        }while( (hti = GetNextSiblingItem( hti )) != NULL );
    }
}
```

6.2.3 收拢所有分支

`CollapseBranch` 函数将某个分支下的条目全部收拢。读者可能有过这样的经历:将所有的分支打开后,满屏幕都是条目,这使得定位某个特定条目非常困难。而清单 6-3 所示的 `CollapseAll` 函数能够收拢所有分支:

清单 6-3 CollapseAll() 函数

```
void CTreeCtrlEX::CollapseAll()
```

```

    {
        HTREEITEM hti = GetRootItem();
        do{
            CollapseBranch( hti );
        }while( (hti = GetNextSiblingItem( hti )) != NULL );
    }

```

在函数中首先得到根条目,然后遍历所有与其同级的分支,并对每个分支调用 CollapseBranch 函数。

6.2.4 拷贝条目

将条目拷贝到新位置十分简单,之所以将这个操作封装到定制的函数中是为了操作的扩展性和易用性。在函数的末尾调用 OnItemCopied 函数,以使派生树视图控件类 (CTreeCtrlEX)有机会更新内部信息。清单 6-4 所示为 CopyItem 的源代码,其中 hItem 指定将被拷贝的条目句柄,htiNewParent 指定了新条目的父条目,htiAfter 指定了将在其后创建新条目的条目。

清单 6-4 CopyItem()函数

```

HTREEITEM CTreeCtrlEX::CopyItem( HTREEITEM hItem, HTREEITEM,
    htiNewParent, HTREEITEM htiAfter /* = TVI_LAST */ )
{
    TV_INSERTSTRUCT    tvstruct;
    HTREEITEM          hNewItem;
    CString            sText;
    // 得到源条目的信息
    tvstruct.item.hItem = hItem;
    tvstruct.item.mask = TVIF_CHILDREN|TVIF_HANDLE| TVIF_IMAGE|
        TVIF_SELECTEDIMAGE;
    GetItem(&tvstruct.item);
    sText = GetItemText( hItem );
    tvstruct.item.cchTextMax = sText.GetLength();
    tvstruct.item.pszText = sText.LockBuffer();
    // 将条目插入到合适的位置
    tvstruct.hParent = htiNewParent;
    tvstruct.hInsertAfter = htiAfter;
    tvstruct.item.mask = TVIF_IMAGE|TVIF_SELECTEDIMAGE|TVIF_TEXT;
    hNewItem = InsertItem(&tvstruct);
    sText.ReleaseBuffer();
    // 限制拷贝条目数据和条目状态
    SetItemData( hNewItem, GetItemData( hItem ));
    SetItemState( hNewItem, GetItemState( hItem, TVIS_STATEIMAGEMASK ),
        TVIS_STATEIMAGEMASK );
    // 调用虚函数以允许派生类进行进一步处理
    OnItemCopied( hItem, hNewItem );
    return hNewItem;
}

```

```

}
void CTreeCtrlEX::OnItemCopied(HTREEITEM /* hItem */, HTREEITEM /* hNewItem */)
{ // Virtual function }

```

在类定义中添加如下声明:

```

public:
    HTREEITEM CopyItem( HTREEITEM hItem, HTREEITEM htiNewParent,
        HTREEITEM htiAfter = TVI_LAST );
protected:
    virtual void OnItemCopied( HTREEITEM hItem, HTREEITEM hNewItem );

```

6.2.5 拷贝分支

使用递归和 CopyItem 函数就可以完成分支拷贝任务,下面使用 CopyBranch 函数封装这一操作。清单 6-5 所示为 CopyBranch 函数的源代码,其中 htiBranch 指定了分支的起始节点,htiNewParent 则指定了新分支的父条目句柄,htiAfter 指定了将在其后创建新分支的条目。

清单 6-5 CopyBranch() 函数

```

HTREEITEM CTreeCtrlEX::CopyBranch( HTREEITEM htiBranch, HTREEITEM htiNewParent,
    HTREEITEM htiAfter /* = TVI_LAST */ )
{
    HTREEITEM hChild;

    HTREEITEM hNewItem = CopyItem( htiBranch, htiNewParent, htiAfter );
    hChild = GetChildItem(htiBranch);
    while( hChild != NULL )
    {
        // 递归以拷贝所有条目
        CopyBranch(hChild, hNewItem);
        hChild = GetNextSiblingItem( hChild );
    }
    return hNewItem;
}

```

6.2.6 移动条目或分支

移动条目或分支很简单,只要在拷贝条目或分支后将源删除即可。也就是说移动条目,只需在调用 CopyItem 函数后调用 DeleteItem。而移动分支,只需在调用 CopyBranch 后调用 DeleteItem(如果删除了一个条目,则其下所有子条目都会被删除)。

6.2.7 得到分支中的最后一个条目

在编程中经常需要得到分支中的最后一个条目,清单 6-6 所示就是封装了此操作的

GetLastItem 函数的源代码,其中 hItem 指定了分支条目,如果为 NULL 则返回大纲中的最后一个条目。

清单 6-6 GetLastItem() 函数

```
HTREEITEM CTreeCtrlEX::GetLastItem( HTREEITEM hItem )
{
    HTREEITEM htiNext;

    if( hItem == NULL ){
        // 得到顶层条目的最后一个
        htiNext = GetRootItem();
        while( htiNext ){
            hItem = htiNext;
            htiNext = GetNextSiblingItem( htiNext );
        }
    }

    while( ItemHasChildren( hItem ) ){
        htiNext = GetChildItem( hItem );
        while( htiNext ){
            hItem = htiNext;
            htiNext = GetNextSiblingItem( htiNext );
        }
    }

    return hItem;
}
```

6.2.8 得到控件中的下一个条目

当树视图控件中的所有分支被完全打开后,按下键盘的下箭头键,则可使当前被选条目的下一个条目处于选中状态。然而如果控件中的分支并没有被完全打开,那么就无法做到这一点。显然,使用箭头键的顺序选择是一个非常有用的特性,它为用户提供了很大的方便,但是其应用会受到当前控件中分支展开状态的影响。我们下面要作的工作就是使这个操作能够忽略放置展开状态的影响。

CTreeCtrl 类中虽然提供了 GetNextItem 函数,但是它并不能直接实现上述操作,因此需要对其进行修改。下面是在 CTreeCtrl 派生类中的函数声明:

```
HTREEITEM GetNextItem( HTREEITEM hItem, UINT nCode ){
    return CTreeCtrl::GetNextItem( hItem, nCode );
}

HTREEITEM GetNextItem( HTREEITEM hItem);
```

可以看到在类中声明了两个版本的 GetNextItem 函数。具有两个参数的函数版本实际就是 CTreeCtrl 的默认实现形式。提供对默认形式的重载是非常必要的,否则新的函数版本将取代旧的版本,从而使旧版本无法再被使用。清单 6-7 所示为新版本 GetNextItem 函数的源代码,其中 hItem 为参考条目句柄:

清单 6-7 GetNextItem() 函数

```

HTREEITEM CTreeCtrlEX::GetNextItem( HTREEITEM hItem )
{
    HTREEITEM    hti;

    if( ItemHasChildren( hItem ) )
        return GetChildItem( hItem );    // 返回第一个子条目
    else {
        // 返回下一个同级条目
        // 如果需要,返回以寻找同级条目
        while((hti = GetNextSiblingItem( hItem )) == NULL ){
            if((hItem = GetParentItem( hItem )) == NULL )
                return NULL;
        }
    }
    return hti;
}

```

6.2.9 得到控件中的上一个条目

上面我们修改 GetNextItem 函数使之能够得到控件中的下一个条目,而无论控件中的分支是否被完全展开。同样的道理,这里将对 GetPrevItem 函数进行重载,其源代码如清单 6-8 所示:

清单 6-8 GetPrevItem() 函数

```

HTREEITEM CTreeCtrlEX::GetPrevItem( HTREEITEM hItem )
{
    HTREEITEM    hti;

    hti = GetPrevSiblingItem(hItem);
    if( hti == NULL )
        hti = GetParentItem(hItem);
    else
        hti = GetLastItem(hti);
    return hti;
}

```

6.3 条目图像编程

树视图控件中的每个条目都可以与一对位图图像相关。这些图像在条目标签的左边显示。其中一张在条目被选择时显示,而另一张则当条目未被选中时显示。例如,当文件夹条目被选中时显示打开的文件夹图像,而在未被选中时则显示关闭的文件夹图像。

将图像索引指定为 L_IMAGECALLBACK 值,可以直到条目将被重绘时再指定其选中或非选中图像。这时控件将直接发送 TVN_GETDISPINFO 通告消息向应用程序请求

索引。

调用 `GetImageList` 成员函数,能够得到树视图控件的图像列表句柄。当需要添加更多图像时,该函数是非常有用的。


6.3.1 设置条目图像

树视图控件为其中的每个条目维护两个图像。其中第一个图像是作为条目图像,显示在条目标签的左侧。而第二个图像则作为状态图像,显示在条目图像的左侧。

默认条件下,所有的条目将在选中和未选中状态下都使用列表中的第一个图像。在 `InsertItem` 函数调用时,通过为条目指定选中态和未选中态图像索引,可以改变这一默认行为。而在添加了某个条目后,还可以通过 `SetItemImage` 函数改变其使用的位图。

调用 `SetImageList` 函数能够设置条目图像。而包含条目图像的图像列表可以是基于包含所有图像的位图创建,也可以是通过添加单个图标完成创建。相对来说,使用位图是一种比较容易的方法。

(1) 创建位图

在资源编辑器中,添加一个包括所有图标的位图资源,例如 。在此位图中的每个图标都是 13x13 像素大小,不过用户可以选择其他的尺寸,不过必须为正方形。

(2) 添加用于存放图像列表的成员变量

该成员变量通常添加到显示控件的窗口类中,例如 `CView` 派生类或 `CDialog` 派生类。

```
public:  
    CImageList m_image;
```

(3) 创建并设置图像列表

调用 `CImageList::Create` 函数,将位图的 ID 作为第一个参数。图标的高度被设置为位图的高度,并做为第二个参数。而第三个参数被设置为新图像列表中的图标数目。既然我们使用位图创建了图像列表,那么一般就不需要动态添加其他图像。因此,将该参数设置为 1。`Create` 函数的最后一个参数指定了掩码颜色。也就是说,使用此颜色的所有像素都将被作为透明色。由于正常的窗口颜色为白色,因此将该参数设置为白色。

当图像列表被创建后,就可以调用 `SetImageList` 将其与树视图控件相联系了。下面给出的示范代码通常应该在 `OnInitDialog` 或 `OnInitialUpdate` 函数中。

```
m_tree.m_image.Create( IDB_OUTLINE, 13, 1, RGB(255,255,255) );  
m_tree.SetImageList( &(m_tree.m_image), TVSIL_NORMAL );
```

(4) 指定条目图标


在为树视图控件设置了图像后,它就能够显示在条目标签的左边了。其所占控件区域与图像列表中相应图标的尺寸相匹配。当向控件中插入新条目时,可以指定该条目将使用的图标,当然也可以在以后的任意时间指定。`InsertItem` 和 `SetItemImage` 函数都需要两个不同的图像值。其中一个用于在条目被选中时显示,而另一个用于在条目未被选中时显示。这两个值可以相同。

6.3.2 设置状态图像

如果指定了状态图像列表,则控件将在条目的左边为状态图像留出空间。应用程序能够使用状态图像,例如使用选中态和清除态复选框来表示应用程序定义的条目状态。条目状态的 12~15 位指定了以 1 为基的状态图像索引(0 表示没有状态图像)。

树视图控件能够为每个条目显示两个图像,这两个图像分属于不同的图像列表。虽然每个条目最多能够拥有 256 个条目图像,但是其状态图像最多只能有 15 个。然而,设置状态图像和设置条目图像的编程是相似的。同样,这里我们也使用位图来创建状态图像。

(1) 创建位图。

在资源编辑器中添加一个位图资源,其中包含所有的状态图像: 。位图中的每个图标都是 13x13 像素大小。用户可以选择其他的尺寸,不过必须为正方形。

(2) 添加用于存放图像列表的成员变量。

该成员变量通常添加到显示控件的窗口类中,例如 CView 派生类或 CDialog 派生类。

```
CImageList m_imageState;
```

(3) 创建并设置图像列表。

调用 CImageList::Create 函数,创建图像列表。然后调用 SetImageList 将其与树视图控件相联系,注意其中使用 TVSIL_STATE 标志。

```
m_tree.m_imageState.Create( IDB_STATE, 13, 1, RGB(255,255,255) );
m_tree.SetImageList( &(m_tree.m_imageState), TVSIL_STATE );
```

(4) 指定状态图标,示范代码如下:

```
CString str = "xyzASDFqwerZCV";
TV_INSERTSTRUCT tv_is;
tv_is.hParent = parent ? parent : TVI_ROOT;
tv_is.hInsertAfter = TVI_LAST;
tv_is.item.mask = TVIF_TEXT | TVIF_STATE;
tv_is.item.stateMask = TVIS_STATEIMAGEMASK;
tv_is.item.state = INDEXTOSTATEIMAGEMASK( 1 );
tv_is.item.pszText = str.GetBuffer(1);
tv_is.item.cchTextMax = str.GetLength();
hItem = InsertItem( &tv_is );
str.ReleaseBuffer();
SetItemState( hItem, INDEXTOSTATEIMAGEMASK(1), TVIS_STATEIMAGEMASK );
```

6.3.3 使用覆盖图像

树视图控件的图像列表中可以包含覆盖图像。条目状态的 8~11 位指定了覆盖图像

索引(0 表示没有覆盖图像)。由于使用的是以 1 为基的 4 位索引,因此覆盖图像的索引一定小于 16。

(1) 设置覆盖图像

调用 `SetOverlayImage` 函数设置覆盖图像,函数使用的图像列表对象就是包含控件条目图像的图像列表对象。该函数的调用可以在图像列表对象与控件相关联之前,也可以在两者相关联之后。一般来说,设置覆盖图像与设置图像列表的操作是一起进行的。

如果只使用一个图像作为覆盖图像,则可以选中 4 个索引中的任意一个(1~4)。而且,如果对两个不同的图像调用 `SetOverlayImage` 函数时使用了相同的覆盖图像索引,那么后面的调用将取代前面的调用。示范代码如下:

```
m_tree.m_image.SetOverlayImage(15, 1 );
m_tree.m_image.SetOverlayImage(16, 2 );
m_tree.m_image.SetOverlayImage(18, 3 );
m_tree.m_image.SetOverlayImage(20, 4 );
```

(2) 将条目状态设置为使用覆盖图像

设置覆盖图像可以在插入新条目时进行,也可以在以后通过调用 `SetItemState` 完成。示范代码如下:

```
// 使用第三个覆盖图像
SetItemState( hItem, INDEXTOOVERLAYMASK(3), TVIS_OVERLAYMASK );

// 清除覆盖图像
SetItemState( hItem, INDEXTOOVERLAYMASK(0), TVIS_OVERLAYMASK );
```

6.4 条目检索操作编程

6.4.1 检索匹配标签

树视图控件并没有提供在条目标签中寻找匹配字符串的功能。本节中将完成这一功能的设计。

(1) 在类中添加两个成员函数

其中 `FindItem` 函数负责完成检索操作,而 `IsFindValid` 函数由 `FindItem` 函数调用,负责对检索结果进行过滤,保留有效条目。

```
public:
    virtual HTREEITEM FindItem(CString &sSearch, BOOL bCaseSensitive = FALSE,
        BOOL bDownDir = TRUE, BOOL bWholeWord = FALSE, HTREEITEM hItem = NULL);
protected:
    virtual BOOL IsFindValid( HTREEITEM );
```

(2) 设计 `FindItem` 函数

在 `FindItem` 函数中调用先前设计的 `GetNextItem`、`GetPrevItem` 和 `GetLastItem` 函数遍历条

目,比较条件字符串与条目标签,以寻找匹配条目。函数执行完毕后,返回匹配条目的句柄,如未找到匹配条目则返回 NULL。该函数的源代码如清单 6-9 所示,其中 str 指定了条件字符串,bCaseSensitive 指定了检索是否要考虑大小写,bDownDir 指定了检索方向(为 TRUE,则向前检索),bWholeWord 指定了是否需要匹配整个字符串,hItem 指定了检索的起始位置。

清单 6-9 GetItem()函数

```
HTREEITEM CTreeCtrlEX::FindItem(CString &str, BOOL bCaseSensitive,
                                BOOL bDownDir, BOOL bWholeWord, HTREEITEM hItem)
{
    int lenSearchStr = str.GetLength();
    if( lenSearchStr == 0 ) return NULL;

    HTREEITEM htiSel = hItem ? hItem : GetSelectedItem();
    HTREEITEM htiCur = bDownDir ? GetNextItem( htiSel ) : GetPrevItem( htiSel );
    CString sSearch = str;

    if( htiCur == NULL )
    {
        if( bDownDir ) htiCur = GetRootItem();
        else htiCur = GetLastItem( NULL );
    }

    if( ! bCaseSensitive )
        sSearch.MakeLower();

    while( htiCur && htiCur != htiSel )
    {
        CString sItemText = GetItemText( htiCur );
        if( ! bCaseSensitive )
            sItemText.MakeLower();

        int n;
        while( (n = sItemText.Find( sSearch )) != -1 )
        {
            // 检索找到的字符串
            if( bWholeWord )
            {
                // 检查前一个字符
                if( n != 0 )
                {
                    if( isalpha(sItemText[n-1]) ||
                        sItemText[n-1] == '_' ) {
                        // Not whole word
                        sItemText = sItemText.Right(
                            sItemText.GetLength() - n -
                            lenSearchStr );
                        continue;
                    }
                }
            }
        }
    }
}
```

```

// 检查下一个字符
if( sItemText.GetLength() > n + lenSearchStr
    && ( isalpha(sItemText[n + lenSearchStr]) ||
        sItemText[n + lenSearchStr] == '_' ) )
{
    // 如果不是全字匹配
    sItemText = sItemText.Right( sItemText.GetLength()
        - n - sSearch.GetLength() );
    continue;
}

if( IsFindValid( htiCur ) )
    return htiCur;
else break;
}

htiCur = bDownDir ? GetNextItem( htiCur ) : GetPrevItem( htiCur );
if( htiCur == NULL )
{
    if( bDownDir ) htiCur = GetRootItem();
    else htiCur = GetLastItem( NULL );
}

return NULL;
}

```

在上述代码中,对匹配条目还要调用 IsFindValid 函数,以判断其是否有效。设计 IsFindValid 函数的目的是为了给用户定制过滤的方式。默认情况下,该函数返回 TRUE。清单 6-10 所示为 IsFindValid 函数的源代码:

清单 6-10 IsFindValid() 函数

```

BOOL CTreeCtrlEX::IsFindValid( HTREEITEM )
{
    return TRUE;
}

```

6.4.2 检索匹配数据

与 FindItem 类似,FindItemData 也是用于检索匹配条目的。只是其检索条件不是字符串,而是与条目相关的数据。如果找到匹配的条目,则返回该条目句柄,否则返回 NULL。当然,执行此函数的控件必须有通过 SetItemData 函数设置的数据。此外,条目掩码中必须包括 TVIF_PARAM。清单 6-11 所示为 FindItemData 函数的源代码,其中 lparam 指定将检索的数据,bDownDir 指定了检索方向,hItem 指定了检索的起始位置。

清单 6-11 FindItemData() 函数

```

HTREEITEM CTreeCtrlEX::FindItemData( DWORD lparam, BOOL bDownDir, HTREEITEM
hItem)

```



```

{
    HTREEITEM htiSel = hItem ? hItem : GetSelectedItem();
    HTREEITEM htiCur = bDownDir ? GetNextItem( htiSel ) :
    GetPrevItem( htiSel );
    if( htiCur == NULL )
    {
        if( bDownDir )
            htiCur = GetRootItem();
        else
            htiCur = GetLastItem( NULL );
    }
    while( htiCur && htiCur != htiSel )
    {
        DWORD sItemData = GetItemData( htiCur );
        if( sItemData == lParam )
            return htiCur;
        htiCur = bDownDir ? GetNextItem( htiCur ) : GetPrevItem( htiCur );
        if( htiCur == NULL )
        {
            if( bDownDir )
                htiCur = GetRootItem();
            else
                htiCur = GetLastItem( NULL );
        }
    }
    return NULL;
}

```

6.4.3 检索匹配 TV_ITEM 结构

FindNextItem 函数将遍历树视图控件中的所有条目,寻找匹配 TV_ITEM 结构中所有属性设置的条目。在 TV_ITEM 结构中,mask 成员指定了那些属性组成了检索条件。如果找到匹配条目,则函数返回该条目的句柄,否则返回 NULL。在 FindNextItem 函数中使用了前文设计的 GetNextItem 函数。如果将函数的 hItem 指定为 NULL,则检索将从根条目开始。清单 6-12 所示为 FindNextItem 函数的源代码:

清单 6-12 FindNextItem() 函数

```

HTREEITEM CTreeCtrlEx::FindNextItem(TV_ITEM* pItem, HTREEITEM hItem)
{
    ASSERT(::IsWindow(m_hWnd));
    TV_ITEM hNextItem;
    //清除条目数据
    ZeroMemory(&hNextItem, sizeof(hNextItem));

    //mask 成员用于得到那些数据应该被比较
    hNextItem.mask = pItem->mask;
    hNextItem.hItem = (hItem) ? GetNextItem(hItem) : GetRootItem();
}

```

```

//准备比较 pszText
if((pItem->mask & TVIF_TEXT) && pItem->pszText)
{
    hNextItem.cchTextMax = strlen(pItem->pszText);
    if(hNextItem.cchTextMax)
        hNextItem.pszText = new char[ ++ hNextItem.cchTextMax];
}

while(hNextItem.hItem)
{
    if(Compare(pItem, hNextItem))
    {
        //将所有信息拷贝到 pItem 中然后返回
        memcpy(pItem, &hNextItem, sizeof(TV_ITEM));
        if(hNextItem.pszText)
            delete hNextItem.pszText;
        return pItem->hItem;
    }

    //在调用 Compare 之前,得到所有需要比较的数据
    hNextItem.mask = pItem->mask;
    hNextItem.hItem = GetNextItem(hNextItem.hItem);
}

pItem->hItem = NULL;
if(hNextItem.pszText)
    delete hNextItem.pszText;
return NULL;
}

```

在 FindNextItem 函数中,通过 TV_ITEM 结构的 mask 成员确定将进行比较的数据。然后调用 Compare 函数执行比较操作,清单 6-13 所示为 Compare 函数的源代码:

清单 6-13 Compare()函数

```

BOOL CTreeCtrlEx::Compare(TV_ITEM* pItem, TV_ITEM& tvTempItem)
{
    GetItem(&tvTempItem);

    //重新设置 mask 以跟踪匹配属性
    tvTempItem.mask = 0;

    if((pItem->mask & TVIF_STATE) &&
        (pItem->state == tvTempItem.state))
        tvTempItem.mask |= TVIF_STATE;

    if((pItem->mask & TVIF_IMAGE) &&
        (pItem->iImage == tvTempItem.iImage))
        tvTempItem.mask |= TVIF_IMAGE;

    if((pItem->mask & TVIF_PARAM) &&
        (pItem->lParam == tvTempItem.lParam))
        tvTempItem.mask |= TVIF_PARAM;
}

```

```

    if((pItem->mask & TVIF_TEXT) &&
        pItem->pszText && tvTempItem.pszText && //Don't compare if either is
        NULL
        ! strcmp(pItem->pszText, tvTempItem.pszText))
        tvTempItem.mask |= TVIF_TEXT;

    if((pItem->mask & TVIF_CHILDREN) &&
        (pItem->cChildren == tvTempItem.cChildren))
        tvTempItem.mask |= TVIF_CHILDREN;

    if((pItem->mask & TVIF_SELECTEDIMAGE) &&
        (pItem->iSelectedImage == tvTempItem.iSelectedImage))
        tvTempItem.mask |= TVIF_SELECTEDIMAGE;

    //如果此时两个值还是同样的,那么它们就匹配
    return (pItem->mask == tvTempItem.mask);
}

```

6.5 编辑条目标签

当将条目添加到树视图控件中时,一般需要指定条目的标签文本。InsertItem 成员函数将 TVITEM 结构作为参数,该参数中就包括了条目的标签信息。树视图控件会为条目排序分配内存,其中条目文本占据了相当大的部分。如果应用程序负责维护字符串的拷贝,也就是说将 TV_ITEM 结构的 pszText 成员或 lpszItem 参数指定为 LPSTR_TEXTCALLBACK,而不是直接传递字符串,那么就会降低控件对内存的需求。使用 LPSTR_TEXTCALLBACK 的控件将在需要重绘时,通过应用程序得到条目的标签。这时,树视图控件会发送 TVN_GETDISPINFO 通告消息,而在应用程序中也需要对该通告进行响应,并在响应函数中设置 NMTVDISPINFO 结构的成员。

树视图控件使用的内存是由创建控件的进度分配的,因此控件允许的最大条目数取决于堆上的可用内存大小,其中每个条目需要 64 字节的内存。

6.5.1 编辑标签

对于 TVS_EDITLABELS 风格的树视图控件,用户只要单击具有焦点的条目标签,就可以直接对其进行编辑。而应用程序则通过 EditLabel 成员函数编辑条目。无论是对标签开始编辑、取消编辑或完成编辑,树视图控件都会发送通告消息。

当开始编辑标签时,树视图控件会发送 TVN_BEGINLABELEDIT 通告消息。通过对该通告消息的响应,可以允许对某些标签进行编辑或禁止对某些标签进行编辑。如果响应函数返回 0,则允许编辑,否则不允许编辑。

在标签编辑时,一般在 TVN_BEGINLABELEDIT 通告消息响应函数中调用 GetEditControl 函数得到用于编辑标签的编辑控件指针。在函数中还可以调用 SetLimitText 函数限制输入的文本长度,或归类编辑控件使其具有更强的功能(例如只允许输入数字,常见第 4

章)。需要注意的是,编辑控件只是在 TVN_BEGINLABELEDIT 消息发送后才会显示。清单 6-14 所示为 TVN_BEGINLABELEDIT 通告消息的处理函数 OnBeginLabelEdit 的源代码:

清单 6-14 OnBeginLabelEdit() 函数

```
void CTreeCtrlEX::OnBeginLabelEdit(NMHDR * pNMHDR, LRESULT * pResult)
{
    TV_DISPINFO * pTVDispInfo = (TV_DISPINFO *)pNMHDR;

    // 限制文本长度为 127 个字符
    GetEditControl()->LimitText(127);

    *pResult = 0;
}
```

当标签编辑完成或取消时,树视图控件会发送 TVN_ENDLABELEDIT 通告消息。其中的 lParam 参数即为 NMTVDISPINFO 结构的地址,TV_ITEM 则为条目成员,其中包含了被编辑的文本。应用程序应该在响应函数中更新条目标签。如果 TV_ITEM 结构的 pszText 成员为 0,则表示编辑被取消。清单 6-15 所示为 TVN_ENDLABELEDIT 通告的处理函数 OnEndLabelEdit 的源代码:

清单 6-15 OnEndLabelEdit() 函数

```
void CTreeCtrlEX::OnEndLabelEdit(NMHDR * pNMHDR, LRESULT * pResult)
{
    // 设置 pResult 为 TRUE,以接受改变
    *pResult = TRUE;
}
```

6.5.2 使用 Esc 和 Return 键结束编辑

在默认情况下,除非用鼠标单击编辑控件外的区域,否则无法终止编辑操作。本节将向读者介绍如何利用键盘的 Esc 键和 Return 键终止编辑。如果树视图控件的父窗口为 CDialog 或 CFormView,则 Esc 和 Return 键被按下的消息由其父窗口进行处理,而不会抵达控件。显然,如果能将此处理封装到控件管理类中将是最好的。MFC 提供了 PreTranslateMessage 函数,使控件能够在父窗口对消息处理之前截获所需消息并对之进行处理。下面的代码除了对 Esc 键和 Enter 键进行处理外,还处理 Ctrl + C 等组合键。也就是说,用户也能对在位编辑控件进行拷贝和粘贴操作。清单 6-16 所示为 PreTranslateMessage 消息处理函数:

清单 6-16 PreTranslateMessage() 函数

```
BOOL CTreeCtrlEX::PreTranslateMessage(MSG * pMsg)
{
    if( pMsg->message == WM_KEYDOWN )
    {
        if( GetEditControl() && (pMsg->wParam == VK_RETURN
            || pMsg->wParam == VK_DELETE || pMsg->wParam == VK_ESCAPE
```

```

        || GetKeyState( VK_CONTROL)))
    {
        ::TranslateMessage(pMsg);
        ::DispatchMessage(pMsg);
        return TRUE;
    }
}
return CTreeCtrl::PreTranslateMessage(pMsg);
}

```

6.5.3 禁止编辑标签

如果以 TVS_EDITLABELS 风格创建树视图控件,则在默认情况下,用户能够编辑其中任何条目的标签。但是在某些情况下,我们并希望所有的条目都能够被编辑。因此,必须设置一定的条件。而对条目可编辑性的检查也应该在 PreTranslateMessage 函数中完成。清单 6-17 所示为在 PreTranslateMessage 函数中添加的代码:

清单 6-17 PreTranslateMessage() 函数

```

BOOL CTreeCtrlEX::PreTranslateMessage(MSG * pMsg)
{
    ...
    if( pMsg->message == WM_LBUTTONDOWN )
    {
        UINT      flag = TVHT_ONITEMLABEL;
        CPoint     pt = pMsg->pt;

        ScreenToClient(&pt);
        if( HitTest( pt, &flag ) == GetSelectedItem()
            // && 在这里可以检查条目是否能够被编辑)
        {
            SetFocus(); //为树视图控件设置焦点
            return TRUE ;
        }
    }
    ...
    return CTreeCtrl::PreTranslateMessage(pMsg);
}

```

6.5.4 树视图控件状态

树视图控件中的每个条目都有各自的当前状态。例如,某个条目可能被选择、展开、禁止等等。对于大多数条目来说,控件会自动设置状态以反映用户的操作。当然,也可以在应用程序中通过 SetItemState 和 GetItemState 函数来设置和检索条目状态。

当指定或修改条目状态时,nStateMask 参数指定了将被设置的位,nState 参数则指定这些位的新值。下面的示范代码将父条目(hParentItem)的状态改变为 TVIS_EXPANDPAR-

TIAL:

```
TVITEM curItem;
HTREEITEM hParentItem;

hParentItem = m_treeCtrl.GetSelectedItem();

//修改父条目,以保持'+ '号
curItem.mask = TVIF_HANDLE;
curItem.hItem = hParentItem;
m_treeCtrl.GetItem(&curItem);

curItem.mask = TVIF_STATE;
curItem.state = TVIS_EXPANDPARTIAL;
curItem.stateMask = TVIS_EXPANDPARTIAL;
m_treeCtrl.SetItem(&curItem);
```

如果要改变条目的覆盖图像,则需要在 `nStateMask` 中设置 `TVIS_OVERLAYMASK` 位,并且在 `nState` 中使用 `INDEXTOOVERLAYMASK` 宏指定覆盖图像索引。索引值可以为 0,这表示不使用覆盖图像。覆盖图像必须通过 `CImageList::SetOverlayImage` 函数添加到树视图控件中。该函数以 1 为基的图像索引指定了将被添加的图像,此索引是通过 `INDEXTOOVERLAYMASK` 宏得到的。树视图控件最多能够有 4 张覆盖图像。

如果要改变条目的状态图像,则需要在 `nStateMask` 中设置 `TVIS_STATEIMAGEMASK` 位,并且在 `nState` 中使用 `INDEXTOOVERLAYMASK` 宏指定状态图像索引。索引值可以为 0,这表示不使用状态图像。

6.6 树视图控件的拖拽操作

当用户拖动树视图控件中的条目时,控件会发送通告消息。如果用户使用鼠标左键拖动条目,则控件发送 `TVN_BEGINDRAG` 通告;如果用户使用鼠标右键拖动条目,则控件发送 `TVN_BEGINRDRAG` 通告。控件可以通过指定 `TVS_DISABLEDRAHDROP` 风格来阻止发送这些消息。

调用 `CreateDragImage` 函数能够获得用于在拖动操作中显示的图像。树视图控件将基于被拖动的条目标签创建拖动位图。接着,控件会创建一个图像列表,并将上述位图添加到其中,然后返回 `CImageList` 对象的指针。

为了使条目实际完成拖动,还必须添加自己的代码。这涉及使用图像列表的拖动函数,并要处理在拖动开始后发送的 `WM_MOUSEMOVE`、`WM_LBUTTONDOWN`(或 `WM_RBUTTONDOWN`)消息。

如果控件中的条目为拖拽操作的目标,则需要知道鼠标何时会移动到目标条目上方。这可以通过调用 `HitTest` 函数得到,鼠标光标的位置信息包含在 `TVHITTESTINFO` 结构或 `Point` 结构中。当 `HitTest` 函数返回时,返回结构中包含了鼠标与控件的相对位置信息。如果鼠标在树视图控件的某个条目上,则结构中包含了该条目的句柄。调用 `SetItem` 函数并

将其 state 参数设置为 TVIS_DROPHILITED, 可以将某个条目标记为拖拽操作目标。

6.6.1 实现拖拽

本节将向读者介绍在树视图控件中实现拖拽的步骤, 需要注意的是这里处理的是鼠标左键拖拽, 鼠标右键拖拽的实现方法类似, 如果读者有兴趣可以自行完成。

(1) 确定控件风格支持拖拽

如果控件具有 TVS_DISABLEDRAHDROP 风格, 那么它就不会发送 TVN_BEGINDRAG 通告消息, 也就是说无法真正完成数据的移动(拖拽)。因此, 必须确定控件不具有 TVS_DISABLEDRAHDROP 风格。

(2) 在 CTreeCtrlEX 类中声明成员变量

在 CTreeCtrl 的派生类 CTreeCtrlEX 中添加成员函数, 用于标识是否在进行拖拽, 以及标识拖拽条目的句柄和放置位置。其中 m_pDragImage 成员负责在拖拽操作中保存图像列表。

```
protected:
    CImageList *    m_pDragImage;
    BOOL           m_bLDragging;
    HTREEITEM       m_hitemDrag, m_hitemDrop;
```

(3) 添加 TVN_BEGINDRAG 通告消息的处理函数

使用 ClassWizard 添加 TVN_BEGINDRAG 通告消息的处理函数 OnBeginDrag, 其源代码如清单 6-18 所示:

清单 6-18 OnBeginDrag() 函数

```
void CTreeCtrlEX::OnBeginDrag(NMHDR * pNMHDR, LRESULT * pResult)
{
    NM_TREEVIEW * pNMTreeView = (NM_TREEVIEW *)pNMHDR;
    *pResult = 0;

    m_hitemDrag = pNMTreeView->itemNew.hItem;
    m_hitemDrop = NULL;

    m_pDragImage = CreateDragImage(m_hitemDrag); // 得到拖拽所用的图像列表
    // 如果没有, 则 CreateDragImage 返回 NULL
    if( ! m_pDragImage )
        return;

    m_bLDragging = TRUE;
    m_pDragImage->BeginDrag(0, CPoint(-15, -15));
    POINT pt = pNMTreeView->ptDrag;
    ClientToScreen( &pt );
    m_pDragImage->DragEnter(NULL, pt);
    SetCapture();
}
```

函数首先设置成员变量,接着调用 `CreateDragImage` 函数创建拖拽图像(亦即拖拽时随鼠标移动的图像)。该函数创建的图像由条目图像和标签文本组成。

创建了拖拽图像后,调用 `BeginDrag` 函数设置图像的显示。第一个参数指定所用图像,其中 0 表示使用图像列表中的第一个图像。第二个参数指定了拖拽图像绘制位置相对于鼠标位置的偏移。然后调用 `DragEnter` 函数显示拖拽图像,第一个参数为 `NULL` 表示即使鼠标拖动到控件外部,也依然显示拖拽图像。

(4) 添加 `WM_MOUSEMOVE` 消息处理函数用于更新拖拽图像

在该消息处理函数中,需要更新拖拽图像的位置和放置位置。`DragMove` 函数负责移动拖拽图像。如果鼠标在某个控件条目上,则更新放置目标。需要注意的是对 `DragShowNolock` 函数的调用。第一次调用隐藏拖拽图像,以便控件对其进行更新;而第二次调用则将再次显示拖拽图像。当然也可以联合使用 `DragLeave` 和 `DragEnter` 函数,不过上述方法更有效一些。清单 6-19 所示为 `OnMouseMove` 函数的源代码:

清单 6-19 `OnMouseMove()` 函数

```
void CTreeCtrlEX::OnMouseMove(UINT nFlags, CPoint point)
{
    HTREEITEM    hitem;
    UINT         flags;

    if (m_bLDragging)
    {
        POINT pt = point;
        ClientToScreen(&pt);
        CImageList::DragMove(pt);
        if ((hitem = HitTest(point, &flags)) != NULL)
        {
            CImageList::DragShowNolock(FALSE);
            SelectDropTarget(hitem);
            m_hitemDrop = hitem;
            CImageList::DragShowNolock(TRUE);
        }
    }

    CTreeCtrl::OnMouseMove(nFlags, point);
}
```

(5) 添加对 `WM_LBUTTONDOWN` 消息的处理函数用于完成拖拽操作

拖拽操作结束的标志就是用户释放鼠标左键,因此需要在 `WM_LBUTTONDOWN` 的消息处理函数中完成实际的数据转移。即使不是在进行拖拽,用户也会单击和释放鼠标左键,这时显然不能当作拖拽处理。因此函数首先需要判断是否在进行拖拽。

移动条目或分支数据时,首先需要判断移动是否有效。一般情况下,无需调用 `Expand` 函数。然而,如果我们要实现动态载入,那么判断拷贝位置是非常必要的。另外,在 `OnLButtonDown` 函数中,还需要将创建的拖拽图像删除。

清单 6-20 所示为 `OnLButtonDown` 函数的源代码:

清单 6-20 OnLButtonUp()函数

```

void CTreeCtrlEX::OnLButtonUp(UINT nFlags, CPoint point)
{
    CTreeCtrl::OnLButtonUp(nFlags, point);

    if (m_bLDragging)
    {
        m_bLDragging = FALSE;
        CImageList::DragLeave(this);
        CImageList::EndDrag();
        ReleaseCapture();
        delete m_pDragImage;
        // 取消目标的高亮显示
        SelectDropTarget(NULL);

        if( m_hitemDrag == m_hitemDrop )
            return;

        HTREEITEM htiParent = m_hitemDrop;
        while( (htiParent = GetParentItem( htiParent )) != NULL )
        {
            if( htiParent == m_hitemDrag ) return;
        }

        Expand( m_hitemDrop, TVE_EXPAND );
        HTREEITEM htiNew = CopyBranch( m_hitemDrag, m_hitemDrop, TVI_LAST );
        DeleteItem(m_hitemDrag);
        SelectItem( htiNew );
    }
}

```

6.6.2 处理无意拖拽

使用 Windows 的读者可能会有这样的经历:当使用“资源管理器”时,如果在释放鼠标左键前不慎移动了鼠标就有可能导致文件夹/文件的移动。那么有没有可能处理无意拖拽呢?设置时间延迟就是一个解决方法,也就是说当用户按下鼠标左键后(鼠标下有条目),必须在原位置停留一段时间,才能激活拖拽。下面就是其实现步骤:

- (1) 在类中声明成员变量,用以保存用户按下鼠标左键的时间。

```

protected:
    DWORD m_dwDragStart;

```

- (2) 定义延迟常量。

在类中将延迟常量定义为 DRAG_DELAY,并将其值设置为 80,当然也可以设置为其他值。

```

#define DRAG_DELAY 80

```

(3) 添加 WM_LBUTTONDOWN 消息处理函数。

该函数唯一需要定制的就是初始化 m_dwDragStart。其中调用 GetTickCount 函数,以得到 Windows 启动的毫秒数。清单 6-21 所示为 OnLButtonDown 函数的源代码:

清单 6-21 OnLButtonDown() 函数

```
void CTreeCtrlEX::OnLButtonDown(UINT nFlags, CPoint point)
{
    m_dwDragStart = GetTickCount();
    CTreeCtrl::OnLButtonDown(nFlags, point);
}
```

(4) 在 TVN_BEGINDRAG 消息处理函数中添加延迟检验代码。

在该函数的起始部分添加如下代码,用以检验是否达到所需延迟。如未达到,则直接返回不作任何处理。

```
if( (GetTickCount() - m_dwDragStart) < DRAG_DELAY )
    return;
```

6.6.3 使用 Esc 取消拖拽

允许用户拖拽,当然也应该允许用户取消拖拽。这里将 Escape 键作为功能键,如果用户按下了该键,则取消拖拽。这个处理一般在 PreTranslateMessage 函数中完成,其源代码如清单 6-22 所示:

清单 6-22 PreTranslateMessage() 函数

```
BOOL CTreeCtrlEX::PreTranslateMessage(MSG * pMsg)
{
    if( pMsg->message == WM_KEYDOWN && pMsg->wParam == VK_ESCAPE
        && m_bLDragging)
    {
        m_bLDragging = 0;
        CImageList::DragLeave(NULL);
        CImageList::EndDrag();
        ReleaseCapture();
        SelectDropTarget(NULL);
        delete m_pDragImage;
        return TRUE;        // 不再继续处理
    }

    return CTreeCtrl::PreTranslateMessage(pMsg);
}
```

此时还需要做一件事——在 WM_LBUTTONDOWN 消息处理函数中调用 SetFocus 才能保证整个工作的正确性。因为如果控件还没有获得焦点,那么当用户开始拖拽操作时,控件并不会得到焦点这将使它无法得到 WM_KEYDOWN 消息,也就是说按下 Escape 无法取消拖拽。

6.6.4 处理拖拽操作中的滚动问题

当进行拖拽时,拖拽目标并非总是可见的。因此有时需要用户收拢一些条目,这样使被拖拽的条目和目标条目都可见。然而,这样显然对用户要求过多,况且很多专业的应用程序,例如 Word 都会在拖拽时提供自动滚动支持。

本节将实现在树视图拖拽操作时自动滚动。其中采用定时器来允许拖拽轮廓,同时还根据光标与控件边界的距离设置不同的滚动速度。

(1) 在 CTreeCtrlEX 类中添加成员变量

这里添加两个成员变量,其中 `m_nTimerID` 用于保存定时器 ID, `m_timerticks` 用于保存定时器经过的时间。有人也许会再声明两个变量,用于保存将由定时器产生的事件 ID 以及定时器允许的延迟时间。不过,本类将分别使用硬编码值 1 和 75。

```
Protected:
    UINT    m_nTimerID;
    UINT    m_timerticks;
```

(2) 在 TVN_BEGINDRAG 消息处理函数中设置定时器

将下述语句添加到清单 6-18 所示的 `OnBeginDrag` 消息处理函数末尾:

```
m_nTimerID = SetTimer(1, 75, NULL);
```

(3) 添加 WM_TIMER 消息处理函数

使用 ClassWizard 在类中添加 `WM_TIMER` 消息处理函数 `OnTimer()`。该函数首先确定处理的是正确的定时器事件,然后更新拖拽图像位置,并确定是否需要滚动。

滚动条件应该以鼠标与控件上、下边缘的距离为准。也就是说,当鼠标靠近控件上边缘,或在其上,则控件应该向上滚动;而当鼠标靠近控件的下边缘,或在其下,则控件应该向下滚动。滚动速度则根据鼠标的位置确定。代码中使用硬编码数值 6 作为每个速度级之间的差值。每个速度级别对应于 20 个像素的附加滚动。而基础滚动幅度则由控件决定。

读者可能会注意到,在调用 `DragMove` 函数时使用的是屏幕坐标。当用户开始拖拽操作时,如果指定桌面窗口调用 `DragEnter` 函数,则 `DragMove` 要使用屏幕坐标。如果调用 `DragEnter` 函数时指定了树视图控件,那么使用控件的客户坐标。

在进行下一步之前,还需要讨论一下 `GetVisibleCount` 函数。它应该返回当前的可见条目数,而实际上它返回的是最大可见条目数。因此,需要在调用 `SelectDropTarget` 前进行必要的检查。

清单 6-23 所示为 `OnTimer()` 函数的源代码:

清单 6-23 OnTimer() 函数

```
void CTreeCtrlEX::OnTimer(UINT nIDEvent)
{
    if( nIDEvent != m_nTimerID )
```

```

    {
        CTreeCtrl::OnTimer(nIDEvent);
        return;
    }

    m_timerticks++;
    POINT pt;
    GetCursorPos( &pt );
    RECT rect;
    GetClientRect( &rect );
    ClientToScreen( &rect );

    // 注意:使用的是屏幕坐标,因为对 DragEnter 的调用使用的是桌面窗口
    CImageList::DragMove(pt);

    HTREEITEM hitem = GetFirstVisibleItem();

    if( pt.y < rect.top + 10 )
    {
        // 向上滚动
        int slowscroll = 6 - (rect.top + 10 - pt.y) / 20;
        if( 0 == ( m_timerticks % (slowscroll > 0? Slowscroll : 1) ) )
        {
            CImageList::DragShowNolock(FALSE);
            SendMessage( WM_VSCROLL, SB_LINEUP);
            SelectDropTarget(hitem);
            m_hitemDrop = hitem;
            CImageList::DragShowNolock(TRUE);
        }
    }
    else if( pt.y > rect.bottom - 10 )
    {
        // 向下滚动
        int slowscroll = 6 - (pt.y - rect.bottom + 10) / 20;
        if( 0 == ( m_timerticks % (slowscroll > 0? Slowscroll : 1) ) )
        {
            CImageList::DragShowNolock(FALSE);
            SendMessage( WM_VSCROLL, SB_LINEDOWN);
            int nCount = GetVisibleCount();
            for( int i = 0; i < nCount - 1; ++i )
                hitem = GetNextVisibleItem(hitem);
            if( hitem )
                SelectDropTarget(hitem);
            m_hitemDrop = hitem;
            CImageList::DragShowNolock(TRUE);
        }
    }
}

```

(4) 当拖拽完成后,停止定时器

当拖拽完成后,调用 KillTimer(m_nTimerID)停止定时器。如果是取消拖拽,则该操作

应该在 `PreTranslateMessage` 函数中执行;而如果是正常结束,则应该在 `OnLButtonUp` 函数中执行。

6.6.5 在拖拽中保持条目等级

标准拖拽操作将被拖拽的分支以子条目方式添加到目标条目后。而在很多拖拽操作中,需要保持条目的等级。也就是说,原来是父条目拖拽后也保持其父条目的等级,原来是子条目的也需要保持其原来的等级。那么我们怎么判断两个条目的等级高低呢?条目缩进量就是一个很好的参照。

如果被拖动条目和目标条目具有相同的缩进量,那么就将被拖动条目作为目标的同级条目添加到新位置。如果被拖动条目的缩进量大于目标,则被拖动条目将作为目标的子条目添加到新位置。如果拖动条目的缩进量小于目标,则拖放操作被取消。

(1) 在 `CTreeCtrlEX` 类中添加成员变量

```
BOOL m_bKeepIndentLevel;
```

在类的构造函数中初始化该变量为 `FALSE`:

```
m_bKeepIndentLevel = FALSE;
```

当该成员为 `FALSE` 时,拖拽操作将正常进行。为了保持缩进级别,用户可以在应用程序中将该值设置为 `TRUE`。

(2) 修改释放鼠标左键消息处理函数 `OnLButtonUp`

在函数中添加的代码主要用于检查缩进级别,从而决定拖拽操作的行为。清单 6-24 所示为新添加的代码:

清单 6-24 `OnLButtonUp()` 函数

```
void CTreeCtrlEX::OnLButtonUp(UINT nFlags, Cpoint point)
{
    ...
    while ((htiParent = GetParentItem(htiParent)) != NULL)
    {
        if (htiParent == m_hDragItem) return;
    }

    // 检查缩进级别
    HTREEITEM htiPosition = TVI_LAST;
    if (m_bKeepIndentLevel)
    {
        int nDragIndent = GetIndentLevel(m_hDragItem);
        int nDropIndent = GetIndentLevel(m_hDropItem);

        if (nDragIndent == nDropIndent)
        {
            // 以放置条目的同级条目,添加其后
            htiPosition = m_hDropItem;
        }
    }
}
```

```

        m_hDropItem = GetParentItem(m_hDropItem);
    }
    else if (nDragIndent == (nDropIndent + 1))
    {
        // 如果放置到上层缩进,则将其以子条目方式放置
        htiPosition = TVI_LAST;
    }
    else
    {
        // 无效拖拽目标
        return;
    }
}

Expand(m_hDropItem, TVE_EXPAND);

HTREEITEM htiNew = CopyBranch(m_hDragItem, m_hDropItem, htiPosition);
DeleteItem(m_hDragItem);
...
}

```

在上述代码中使用 GetIndentLevel 函数得到缩进级别,该函数的源代码如清单 6-25 所示:

清单 6-25 GetIndentLevel() 函数

```

int CTreeCtrl::GetIndentLevel(HTREEITEM hItem)
{
    int iIndent = 0;
    while ((hItem = GetParentItem(hItem)) != NULL)
    {
        iIndent++;
    }
    return iIndent;
}

```

6.6.6 增强拖拽功能

下面设计的代码可以禁止对特定条目进行拖拽(例如 ClassView 中树视图控件中的根条目)。还可以确定某条目是否是有效的拖拽目标,如果不是则能够提供另一个可选目标(在 ClassView 中拖动类时,只有文件夹能作为拖拽目标)。此外,还在拖拽时使用了不同的光标。

(1) 在 CTreeCtrlEX 类中添加如下成员变量:

```

protected:
    CImageList* m_pDragImage;
    BOOL m_bLDragging;
    HTREEITEM m_hitemDrag, m_hitemDrop;
    HCURSOR m_dropCursor, m_noDropCursor;

```

其中 `m_dropCursor` 和 `m_noDropCursor` 必须初始化为合适的光标。

(2) 修改 `TVN_BEGINDRAG` 消息处理函数 `OnBeginDrag`。

修改后的函数调用了 `IsDropSource` 虚函数,以确定条目是否能够作为拖拽操作源。如果 `IsDropSource` 返回 `FALSE`,则操作被取消。清单 6-26 所示为修改后 `OnBeginDrag` 函数的源代码:

清单 6-26 `OnBeginDrag()` 函数

```
void CTreeCtrlEX::OnBeginDrag(NMHDR * pNMHDR, LRESULT * pResult)
{
    NM_TREEVIEW * pNMTreeView = (NM_TREEVIEW *)pNMHDR;

    *pResult = 0;
    m_hitemDrag = pNMTreeView->itemNew.hItem;
    m_hitemDrop = NULL;
    SelectItem( m_hitemDrag );
    if (! IsDropSource(m_hitemDrag))
        return;

    m_pDragImage = CreateDragImage(m_hitemDrag);
    if( ! m_pDragImage )
        return;

    m_bLDragging = TRUE;
    m_pDragImage->BeginDrag(0, CPoint(15, 15));
    POINT pt = pNMTreeView->ptDrag;
    ClientToScreen( &pt );
    m_pDragImage->DragEnter(NULL, pt);
    SetCapture();
}
```

(3) 设计 `IsDropSource` 函数。

`IsDropSource` 函数的默认形式如清单 6-27 所示,其中认为所有的条目都是可见的。用户在使用 `CTreeCtrlEX` 类时,只要重载 `IsDropSource` 函数,并在其中设置有效拖拽源的条件即可。

清单 6-27 `IsDropSource()` 函数

```
/* virtual */ BOOL CTreeCtrlEX::IsDropSource(HTREEITEM hItem)
{
    return TRUE; // 所有条目都可以作为源
}
```

(4) 修改鼠标移动消息处理函数 `OnMouseMove`。

修改后的函数调用了 `GetDropTarget` 函数以确定光标所在位置处的条目是否是有效的拖拽目标。如果函数返回 `NULL`,则表示条目不是有效的拖拽目标。如果条目可以作为拖拽目标,则函数返回该条目的 `HTREEITEM` 句柄。当然,如果条目不是有效的拖拽目标,还可以返回另外一个有效条目句柄,这是一种更好的选择。另外,其中还根据操作的状态改

变了光标。

清单 6-28 所示为 OnMouseMove 函数的源代码：

清单 6-28 OnMouseMove() 函数

```
void CTreeCtrlEX::OnMouseMove(UINT nFlags, CPoint point)
{
    HTREEITEM hitem;
    UINT flags;

    if (m_bLDragging)
    {
        POINT pt = point;
        ClientToScreen( &pt );
        CImageList::DragMove(pt);
        if ((hitem = HitTest(point, &flags)) != NULL)
        {
            CImageList::DragShowNolock(FALSE);
            m_hitemDrop = GetDropTarget(hitem);
            SelectDropTarget(m_hitemDrop);
            CImageList::DragShowNolock(TRUE);
        }
        else
            m_hitemDrop = NULL;
        if (m_hitemDrop)
            SetCursor(m_dropCursor);
        else
            SetCursor(m_noDropCursor);
    }
    CTreeCtrl::OnMouseMove(nFlags, point);
}
```

(5) 设计 GetDropTarget 虚函数。

清单 6-29 所示为 GetDropTarget 函数的源代码,读者在使用时可以重载它,以定制自己的操作。

清单 6-29 GetDropTarget() 函数

```
/* virtual */ HTREEITEM CTreeCtrlEX::GetDropTarget(HTREEITEM hItem)
{
    if (hItem == m_hitemDrag || hItem == GetParentItem(m_hitemDrag))
        return NULL;
    return hItem;
}
```

6.7 树视图控件与工具提示

工具提示是重要的界面辅助元素之一,它能够帮助用户快速掌握软件的使用方法。

6.7.1 为条目图像添加工具提示

工具提示能够应用于各种各样的屏幕元素：菜单命令、工具按钮、控件条目等等。但是，读者是否曾经看到过条目图像的工具提示？至少，到目前为止笔者从来都没有见到过。我们知道，树视图控件中的条目具有状态图像、条目图像和覆盖图像。这些图像都是有特定含义的，用户了解这些含义有助于更好地使用软件，因此为其添加工具提示并非哗众取宠。在一些应用程序中，整个帮助文件的效果很不好，其重要原因之一就是用户无法理解其中各种图像的含义。

本节将实现条目图像的工具提示。

(1) 允许工具提示

要使窗口能够使用工具提示，只需要调用 `EnableToolTips(TRUE)` 函数即可。一般这一操作都是在 `PreSubclassWindow` 函数中完成的。无论控件的创建方式如何，MFC 总是会调用该函数。而 `OnCreate` 函数则不是如此，只有当使用 `Create` 或 `CreateEx` 函数时，`OnCreate` 才会被触发。这也就是说，如果控件是在资源编辑器中创建的话，`OnCreate` 函数将不会被调用。清单 6-30 所示为 `PreSubclassWindow` 函数的源代码：

清单 6-30 `PreSubclassWindow()` 函数

```
void CTreeCtrlEX::PreSubclassWindow()
{
    CTreeCtrl::PreSubclassWindow();
    EnableToolTips(TRUE);
}
```

(2) 重载 `OnToolHitTest` 函数

MFC 框架将调用 `OnToolHitTest` 函数以确定是否应该在指定位置显示工具提示。如果函数返回非零值，则表示鼠标所在位置下为一个工具，应该为其显示工具提示。确切地说，函数将窗口中不同的工具返回不同的非零值。

在该函数中只需要处理鼠标落在条目图标或状态图标上的情况。当然，读者也许希望同时为控件的其他元素添加工具提示，这在后面的内容中还会有所介绍。函数需要计算图标的边界矩形并确定其所属条目的句柄(ID)。虽然状态图标和条目图标使用的是相同的条目 ID，但是其返回值是不同的，这将导致 MFC 使用不同的工具提示。

虽然可以在 `OnToolHitTest` 函数中直接指定工具提示文本，但是在此函数中附加过多的计算是不明智的。因为在鼠标移动时它将不断被调用，所以应该对 `OnToolHitTest` 函数进行优化。清单 6-31 所示为 `OnToolHitTest` 函数的源代码：

清单 6-31 `OnToolHitTest()` 函数

```
int CTreeCtrlEX::OnToolHitTest(CPoint point, TOOLINFO * pTI) const
{
    RECT rect;

    UINT nFlags;
```

```

HTREEITEM hitem = HitTest( point, &nFlags );
if( nFlags & TVHT_ONITEMICON )
{
    CImageList * pImg = GetImageList( TVSIL_NORMAL );
    IMAGEINFO imageinfo;
    pImg->GetImageInfo( 0, &imageinfo );

    GetItemRect( hitem, &rect, TRUE );
    rect.right = rect.left - 2;
    rect.left -= (imageinfo.rcImage.right + 2);

    pTI->hwnd = m_hWnd;
    pTI->uId = (UINT)hitem;
    pTI->lpszText = LPSTR_TEXTCALLBACK;
    pTI->rect = rect;
    return pTI->uId;
}
else if( nFlags & TVHT_ONITEMSTATEICON )
{
    CImageList * pImg = GetImageList( TVSIL_NORMAL );
    IMAGEINFO imageinfo;
    pImg->GetImageInfo( 0, &imageinfo );

    GetItemRect( hitem, &rect, TRUE );
    rect.right = rect.left - (imageinfo.rcImage.right + 2);

    pImg = GetImageList( TVSIL_STATE );
    rect.left = rect.right - imageinfo.rcImage.right ;

    pTI->hwnd = m_hWnd;
    pTI->uId = (UINT)hitem;
    pTI->lpszText = LPSTR_TEXTCALLBACK;
    pTI->rect = rect;

    // 返回值应该对不同的条目图标返回不同的值
    return pTI->uId * 2;
}
return -1;
}

```

(3) 添加对 TTN_NEEDTEXT 通告消息的响应函数

当需要显示工具提示文本时,工具提示控件就会发送 TTN_NEEDTEXT 通告消息。由于在第(2)步中已经指定了 LPSTR_TEXTCALLBACK 用于处理该通告,而 ClassWizard 不支持该通告,因此必须在消息映射中手动添加如下代码。其中 TTN_NEEDTEXTA 和 TTN_NEEDTEXTW 分别用于 Windows NT(UNICODE)和 Windows 9x(ANSI)。

```

BEGIN_MESSAGE_MAP(CTreeCtrlEX, CTreeCtrl)
    //{{AFX_MSG_MAP(CTreeCtrlEX)
    ...
    //{{AFX_MSG_MAP
    ON_NOTIFY_EX_RANGE(TTN_NEEDTEXTW, 0, 0xFFFF, OnToolTipText)

```

```
ON_NOTIFY_EX_RANGE(TTN_NEEDTEXTA, 0, 0xFFFF, OnToolTipText)
END_MESSAGE_MAP()
```

而在类定义中还需要添加如下通告消息响应函数声明:

```
protected:
    //||AFX_MSG(CTreeCtrlEX)
    ...
    //||AFX_MSG
    afx_msg BOOL OnToolTipText( UINT id, NMHDR * pNMHDR, LRESULT * pResult );
    DECLARE_MESSAGE_MAP()
```

由于 Windows 9x 和 Windows NT 使用不同的编码方式:ANSI 和 UNICODE。因此在处理时也有一定的不同。函数将忽略由内建工具提示控件发送的消息,这是由于已经为它设置了 TTF_IDISHWND 标志,也就是说树视图控件的窗口句柄等于工具提示控件 ID。基于当前的鼠标位置,函数决定是否需要为条目图标或状态图标显示工具提示文本。清单 6-32 所示的 OnToolTipText 函数就将图标索引作为工具提示的文本索引:

清单 6-32 OnToolTipText()函数

```
BOOL CTreeCtrlEX::OnToolTipText( UINT id, NMHDR * pNMHDR, LRESULT * pResult )
{
    // 需要同时处理 ANSI 和 UNICODE 版本的消息
    TOOLTIPTEXTA * pTTTA = (TOOLTIPTEXTA *)pNMHDR;
    TOOLTIPTEXTW * pTTTW = (TOOLTIPTEXTW *)pNMHDR;
    CString strTipText;
    UINT nID = pNMHDR->idFrom;

    // 不需要处理从内建工具提示中传递的消息
    if( nID == (UINT)m_hWnd &&
        (( pNMHDR->code == TTN_NEEDTEXTA && pTTTA->uFlags & TTF_IDISHWND ) ||
        ( pNMHDR->code == TTN_NEEDTEXTW && pTTTW->uFlags & TTF_IDISHWND )))
        return FALSE;

    // 得到鼠标位置
    const MSG * pMessage;
    CPoint pt;
    pMessage = GetCurrentMessage();
    ASSERT( pMessage );
    pt = pMessage->pt;
    ScreenToClient( &pt );

    UINT nFlags;
    HTREEITEM hitem = HitTest( pt, &nFlags );
    if( nFlags & TVHT_ONITEMICON )
    {
        int nImage, nSelImage;
        GetItemImage( (HTREEITEM) nID, nImage, nSelImage );
        strTipText.Format( "图像 : %d", nImage );
    }
    else
```

```

    {
        strTipText.Format( "状态 : %d", GetItemState( (HTREEITEM) nID,
            TVIS_STATEIMAGEMASK ) );
    }

#ifdef _UNICODE
    if (pNMHDR->code == TTN_NEEDTEXTA)
        lstrcpyn(pTTTA->szText, strTipText, 80);
    else
        _mbstowcsz(pTTTW->szText, strTipText, 80);
#else
    if (pNMHDR->code == TTN_NEEDTEXTA)
        _wcstombsz(pTTTA->szText, strTipText, 80);
    else
        lstrcpyn(pTTTW->szText, strTipText, 80);
#endif
    *pResult = 0;
    return TRUE;
}

```

6.7.2 为条目添加工具提示

相对于为条目图像显示工具提示,为条目本身添加工具提示是一件相当简单的事情。假设在 CFormView 的派生类 CMyFormView 中使用树视图控件,则为其添加工具提示的步骤如下所示:

(1) 在视图类中声明控件变量 m_tree。

(2) 决定条目工具提示的存储方式。

此处将使用映射方式存储工具提示,其声明如下:

```
CMap< HTREEITEM, HTREEITEM&, CString, CString& > tooltipMap;
```

(3) 为控件中的条目添加提示映射,例如在调用 InsertItem 函数时:

```
HTREEITEM hItem = m_tree.InsertItem(&tvItem); // tvItem 的类型为 TV_INSERT-
STRUCT
tooltipMap[hItem] = "工具提示文本"; // 设置条目的工具提示文本
```

(4) 在 CMyFormView 类定义中声明工具提示控件:

```
CToolTipCtrl m_tooltip;
```

(5) 在 OnInitialUpdate(在 CDialog 类中则为 OnInitDialog)消息处理函数中初始化工具提示控件。

```

m_tooltip.Create(this);
m_tooltip.Activate(TRUE);
m_tooltip.AddTool(GetDlgItem(IDC_TREE1), "树视图控件工具提示");

```


(6) 重载 CMyFormView 类的 PreTranslateMessage 函数, 以显示工具提示。

清单 6-33 所示为 PreTranslateMessage 函数的源代码:

清单 6-33 PreTranslateMessage() 函数

```

BOOL CMyFormView::PreTranslateMessage(MSG * pMsg)
{
    if(pMsg->message == WM_MOUSEMOVE && pMsg->hwnd == m_tree.m_hWnd)
    {
        CPoint point(LOWORD(pMsg->lParam), HIWORD(pMsg->lParam));
        HTREEITEM hItem = m_tree.HitTest(point);
        if(hItem != NULL)
        {
            CString text = tooltipMap[hItem];
            m_tooltip.UpdateTipText(text, &m_tree);
            m_tooltip.RelayEvent(pMsg);
        }
    }
}

```

6.8 实现多重选择

当树视图控件中的被选中条目发生变化时, 会发送 TVN_SELCHANGING 和 TVN_SELCHANGED 通告消息。这两个通告消息中都有标识改变是由键盘或鼠标引起的标志。此外, 通告消息中还包括了将得到焦点和将失去焦点的条目信息。程序员可以使用这些信息, 来根据条目状态设置条目属性。如果使 TVN_SELCHANGING 消息的响应函数返回 TRUE, 就会阻止选择操作进行, 而如果返回 FALSE, 则允许选择发生改变。应用程序可以调用 SelectItem 函数来改变选择。

树视图控件对多重选择的支持是非常有限的。它提供了用于设置和获取条目选择状态的消息。另一方面, 控件并不允许不选择条目就直接为其设置焦点。而选择条目后再设置焦点会使先前的条目被解除选择。这为实现多重选择带来了困难。本节给出实现方法如下:

(1) 添加用于记录选择集中第一个条目的成员变量

在 CTreeCtrlEX 类定义中添加如下声明, 并在类的构造函数中将其初始化为 NULL:

```

protected:
    HTREEITEM m_hItemFirstSel;

```

(2) 在 WM_LBUTTONDOWN 消息处理函数中添加代码

一般来说, 选择条目都是通过按下鼠标左键触发的, 因此需要在 WM_LBUTTONDOWN 消息处理函数中添加代码以进行多重选择。

当调用 SelectItem 函数, 或当控件响应鼠标左键单击而选择某个条目时, 先前被选择的条目就会被解除选择。因此, 为了达到多重选择的目的, 就需要重新选择先前的被选条

目。此时需要使用 SetItemState 函数。这时,读者也许会问为什么不使用 SelectItem 函数,或让控件处理鼠标单击呢?这是因为 SetItemState 函数并不会为条目设置焦点,而前面两种方式则相反。与列表视图控件不同,树视图控件不能使用副本 LVIS_FOCUSED 结构来设置条目焦点。

接下来还需要讨论拖拽操作。如果控件没有接收到 WM_LBUTTONDOWN 消息,那么它就不会激活拖拽操作。这样,如果希望支持控件的拖拽操作,那么就必须使控件参与到 WM_LBUTTONDOWN 消息的处理中,或者在定制的 WM_MOUSEMOVE 消息处理函数中激活拖拽。在这里我们选择前面一种方式。

WM_LBUTTONDOWN 消息的内建处理,会导致显示先前被选条目的瞬间的解除选择态。而如果我们使用 SetRedraw(FALSE),则会导致整个控件闪烁。为了解决这个问题,可以调用父窗口的 SetRedraw 函数,而这又将导致工具提示的重绘问题。回头看看这些问题,相对来说瞬间解除选择要相对容易解决一些。

单击已经被选择的条目还会激活标签编辑(解除选择集中的某个选择,势必要再次单击它)。要解决这个问题,只要在继续传递消息前暂时解除条目的选择状态即可。

在 OnLButtonDown 函数中需要对三种不同的条件进行检查。首先,检查是否按下了 Control 键。通常单击 Control 键会标记条目的选择状态。在标记前,先确认鼠标的确单击了某个条目。然后确定将被设置焦点的条目状态。该条目将接着失去焦点,并在调用基类的 OnLButtonDown 函数后被设置为合适的状态。

其次,如果用户按下了 Shift 键,则表示要进行块选择。这时需要保存选择块的第一个条目,这将有助于进一步的处理。SelectItems 函数用于遍历控件中的所有条目,并将选择所有块中的条目,而将其他条目解除选择。

第三,处理通常的鼠标左键单击操作(没有同时按下 Control 或 Shift 键)。这是最简单的一种情况,只要使用 CTreeCtrl 类的默认处理即可。

清单 6-34 所示为 OnLButtonDown 函数的源代码:

清单 6-34 OnLButtonDown() 函数

```
void CTreeCtrlEX::OnLButtonDown(UINT nFlags, CPoint point)
{
    // 如果需要按键,则为控件设置焦点。当按下鼠标左键时,控件并不能自动获得焦点
    m_dwDragStart = GetTickCount();

    if(nFlags & MK_CONTROL)
    {
        // Control key is down
        UINT flag;
        HTREEITEM hItem = HitTest( point, &flag );
        if( hItem )
        {
            // 标记选择状态
            UINT uNewSelState =
                GetItemState( hItem, TVIS_SELECTED ) & TVIS_SELECTED ?
                0 : TVIS_SELECTED;
```

```

        // 得到先前被选条目的状态
        HTREEITEM hItemOld = GetSelectedItem();
        UINT uOldSelState = hItemOld ?
            GetItemState(hItemOld, TVIS_SELECTED) : 0;
        // 选择新条目
        if( GetSelectedItem() == hItem )
            SelectItem( NULL );    // 禁止编辑
        CTreeCtrl::OnLButtonDown(nFlags, point);
        // 为新条目设置合适的选择状态(高亮)
        SetItemState(hItem, uNewSelState, TVIS_SELECTED);
        // 恢复被选择条目的状态
        if (hItemOld && hItemOld != hItem)
            SetItemState(hItemOld, uOldSelState, TVIS_SELECTED);
        m_hItemFirstSel = NULL;
        return;
    }
}
else if(nFlags & MK_SHIFT)
{
    // 按下了 Shift 键
    UINT flag;
    HTREEITEM hItem = HitTest( point, &flag );
    // 如果是第一个 shift 选择,则初始化参照条目
    if( ! m_hItemFirstSel )
        m_hItemFirstSel = GetSelectedItem();
    // 选择新条目
    if( GetSelectedItem() == hItem )
        SelectItem( NULL );    // 禁止编辑
    CTreeCtrl::OnLButtonDown(nFlags, point);
    if( m_hItemFirstSel )
    {
        SelectItems( m_hItemFirstSel, hItem );
        return;
    }
}
else
{
    // 正常 —— 清除所有选项,并使用默认方式处理
    ClearSelection();
    m_hItemFirstSel = NULL;
}

CTreeCtrl::OnLButtonDown(nFlags, point);
}

```

(3) 在 WM_KEYDOWN 消息处理函数中添加代码

OnKeyDown 函数将允许用户使用 Shift + 上箭头、Shift + 下箭头组合键进行选择。如果所按的键不是控制字符,则选择将被清除。清单 6-35 所示为 OnKeyDown 函数的源代码:

清单 6-35 OnKeyDown() 函数

```

void CTreeCtrlEX::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    if ( (nChar == VK_UP || nChar == VK_DOWN) && GetKeyState( VK_SHIFT ) & 0x8000 )
    {
        // 如果是第一个被选表明,则初始化参照条目
        if( ! m_hItemFirstSel )
        {
            m_hItemFirstSel = GetSelectedItem();
            ClearSelection();
        }
        // 寻找当前的被选条目
        HTREEITEM hItemPrevSel = GetSelectedItem();
        HTREEITEM hItemNext;
        if (nChar == VK_UP )
            hItemNext = GetPrevVisibleItem( hItemPrevSel );
        else
            hItemNext = GetNextVisibleItem( hItemPrevSel );
        if ( hItemNext )
        {
            // 确定是否需要重新选择先前的被选条目
            BOOL bReselect =
                ! ( GetItemState( hItemNext, TVIS_SELECTED ) & TVIS_SELECTED );
            // 选择下一个条目 这将导致先前条目被解除选择
            SelectItem( hItemNext );
            // 重新选择先前被选择的条目
            if (bReselect )
                SetItemState( hItemPrevSel, TVIS_SELECTED, TVIS_SELECTED );
        }
        return;
    }
    else if( nChar >= VK_SPACE )
    {
        m_hItemFirstSel = NULL;
        ClearSelection();
    }
    CTreeCtrl::OnKeyDown(nChar, nRepCnt, nFlags);
}

```

(4) 设计用于清除选择的辅助函数 ClearSelection

在没有按下 Control 键或 Shift 键的情况下,每次用户单击树视图控件时,该函数都会被调用。ClearSelection 函数的形式非常简单,它遍历控件中的所有条目并一一清除其选择标记。如果控件中的条目过多,函数的执行可能会变得非常缓慢。另外,其中使用的 GetNextItem 函数已经在 6.1.3 节中进行了介绍。清单 6-36 所示为 ClearSelection 函数的源代码:

清单 6-36 ClearSelection() 函数

```

void CTreeCtrlEX::ClearSelection()
{
    for ( HTREEITEM hItem = GetRootItem(); hItem != NULL; hItem = GetNextItem( hItem ) )
        if ( GetItemState( hItem, TVIS_SELECTED ) & TVIS_SELECTED )
            SetItemState( hItem, 0, TVIS_SELECTED );
}

```

(5) 设计选择某个范围中条目的辅助函数 SelectItems

当用户在按下 Shift 键的情况下,单击鼠标左键时被调用。它负责选择某个范围内的条目,并清除范围外的所有被选条目。清单 6-37 所示为 SelectItems 函数的源代码,其中 hItemFrom 为范围的起始位置,hItemTo 为范围的终止位置:

清单 6-37 SelectItems() 函数

```

BOOL CTreeCtrlEX::SelectItems(HTREEITEM hItemFrom, HTREEITEM hItemTo)
{
    HTREEITEM hItem = GetRootItem();
    // 清除起始条目之前的选择
    while ( hItem && hItem != hItemFrom && hItem != hItemTo )
    {
        hItem = GetNextVisibleItem( hItem );
        SetItemState( hItem, 0, TVIS_SELECTED );
    }
    if (! hItem )
        return FALSE;    // 条目不可见
    SelectItem( hItemTo );
    // 重新排列 hItemFrom 和 hItemTo,使 hItemFirst 处于顶端
    if( hItem == hItemTo )
    {
        hItemTo = hItemFrom;
        hItemFrom = hItem;
    }
    // 遍历其余可见条目
    BOOL bSelect = TRUE;
    while ( hItem )
    {
        // 根据条目是否在范围中,解除选择或选择条目
        SetItemState( hItem, bSelect ? TVIS_SELECTED : 0, TVIS_SELECTED );
        // 是否需要开始从选择中清除条目
        if( hItem == hItemTo )
            bSelect = FALSE;
        hItem = GetNextVisibleItem( hItem );
    }
    return TRUE;
}

```

(6) 设计其他辅助函数

在类中还需要设计其他一些辅助函数用于封装常用的操作。例如,得到第一个被选

条目(GetFirstSelectedItem),得到下一个被选条目(GetNextSelectedItem)以及得到上一个被选条目等(GetPrevSelectedItem)。它们的源代码分别如清单 6-38、6-39 和 6-40 所示:

清单 6-38 GetFirstSelectedItem()函数

```
HTREEITEM CTreeCtrlEX::GetFirstSelectedItem()
{
    for ( HTREEITEM hItem = GetRootItem(); hItem!=NULL; hItem = GetNextItem( hItem ) )
        if (GetItemState( hItem, TVIS_SELECTED ) & TVIS_SELECTED )
            return hItem;
    return NULL;
}
```

清单 6-39 GetNextSelectedItem()函数

```
HTREEITEM CTreeCtrlEX::GetNextSelectedItem( HTREEITEM hItem )
{
    for ( hItem = GetNextItem( hItem ); hItem!=NULL; hItem = GetNextItem( hItem ) )
        if (GetItemState( hItem, TVIS_SELECTED ) & TVIS_SELECTED )
            return hItem;
    return NULL;
}
```

清单 6-40 GetPrevSelectedItem()函数

```
HTREEITEM CTreeCtrlEX::GetPrevSelectedItem( HTREEITEM hItem )
{
    for ( hItem = GetPrevItem( hItem ); hItem!=NULL; hItem = GetPrevItem( hItem ) )
        if (GetItemState( hItem, TVIS_SELECTED ) & TVIS_SELECTED )
            return hItem;
    return NULL;
}
```

6.9 改善条目形式和外观

树视图控件由控件窗口和条目组成,因此改善控件的外观也要从这两方面入手。本节中主要向读者介绍如何改善控件条目。

6.9.1 鼠标敏感条目

在“资源管理器”中进行拖拽时,当鼠标停留在某个节点上一段时间后,该节点会自动展开。本节就将实现这一功能。

(1) 在 CTreeCtrlEX 类中添加两个成员变量

添加的变量中,m_nHoverTimerID 为敏感定时器 ID,m_HoverPoint 则为鼠标位置。当鼠标在拖拽中停止在某个节点上时,敏感定时器将被启动。m_HoverPoint 将保存当前的鼠标位置。

```
private:
    UINT      m_nHoverTimerID;
    POINT      m_HoverPoint;
```

(2) 修改鼠标移动响应函数 OnMouseMove

对该函数的修改主要用于检查敏感定时器是否存在,如果存在则将其删除。因为如果鼠标在移动,那么就无需开启敏感定时器。需要添加的代码如清单 6-41 所示:

清单 6-41 OnMouseMove()函数

```
void CTreeCtrlEX::OnMouseMove(UINT nFlags, CPoint point)
{
    HTREEITEM      hitem;
    UINT           flags;

    if ( m_nHoverTimerID )
    {
        KillTimer( m_nHoverTimerID );
        m_nHoverTimerID = 0;
    }

    if (m_bDragging)
    {
        ...
    }
    ...
}
```

(3) 添加 OnTimer 函数

OnTimer 事件在鼠标停留一定时间后被调用。SetTimer 函数将创建定时器,并在超过停留时间后发送消息以触发 OnTimer 事件。OnTimer 事件处理函数首先清除定时器,然后得到当前鼠标下的条目,并展开该条目。清单 6-42 所示为 OnTimer 函数的源代码:

清单 6-42 OnTimer()函数

```
void CTreeCtrlEX::OnTimer(UINT nIDEvent)
{
    if ( nIDEvent == m_nHoverTimerID )
    {
        KillTimer( m_nHoverTimerID );
        m_nHoverTimerID = 0;
        HTREEITEM trItem = 0;
        uint uFlag = 0;
        trItem = HitTest(m_HoverPoint, &uFlag);
        if ( trItem )
        {
            SelectItem( trItem );
            Expand(trItem,TVE_EXPAND);
        }
    }
}
```

```


else
{
    CTreeCtrl::OnTimer(nIDEvent);
}
}

```

6.9.2 为条目添加复选框

本节将向读者介绍如何创建具有复选框的树视图控件,如图 6-2 所示。

(1) 创建具有复选框的图像

在资源编辑器中插入新的位图资源 IDB_STATE,其图像为: 。位图中包含 5 个图标,每个图标的尺寸为 13x13 像素。由于位图将用于状态图像列表,因此第一个图标为空(状态图标索引从 1 开始),第二个图标标识未复选条目,第三个图标标识复选条目,第四个图标标识未复选条目,但同时表示它至少有一个子条目被复选。第五个图标标识条目被复选,并且至少其一个子条目也被复选。

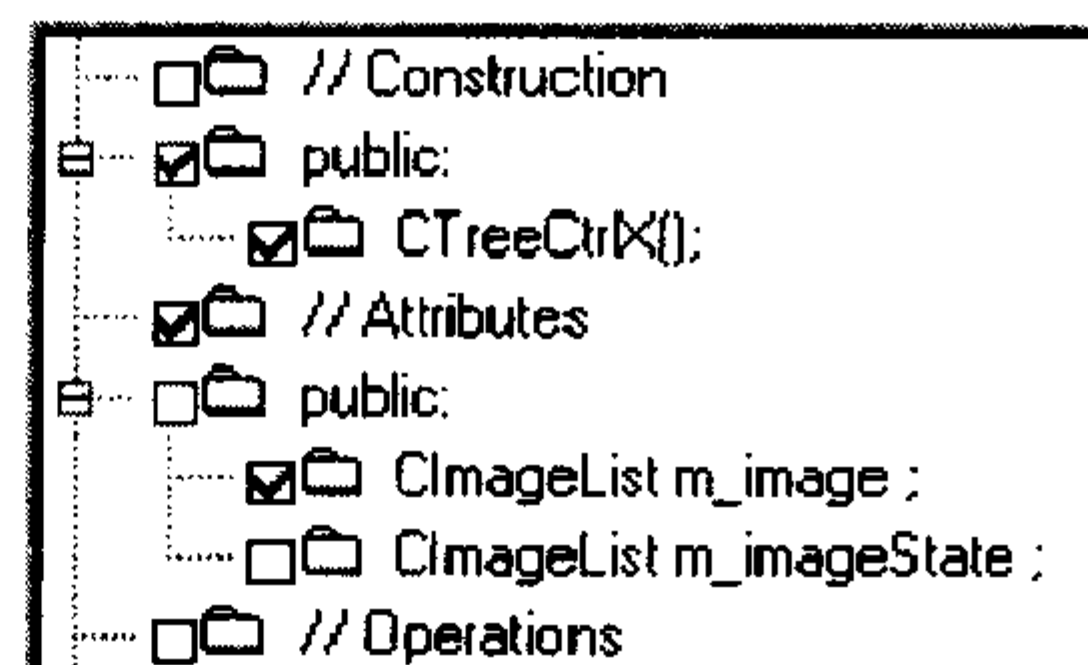


图 6-2 具有复选框的树视图控件

(2) 初始化状态图像列表

下面的代码使用 IDB_STATE 创建图像列表,并将其与树视图控件相联系:

```

m_tree.m_imageState.Create( IDB_STATE, 13, 1, RGB(255,255,255) );
m_tree.SetImageList( &(m_tree.m_imageState), TVSIL_STATE );

```

(3) 插入带有复选状态图像的条目

如果使用 TV_INSERTSTRUCT 结构来执行条目插入操作,则应该指定其 state 和 stateMask 成员。状态图标的索引由 INEXTOSTATEIMAGEMASK 宏确定。当然,也可以调用 SetItemState 函数设置状态图像。示范代码如下:

```

SetItemState( hItem, INEXTOSTATEIMAGEMASK(1), TVIS_STATEIMAGEMASK );

```

(4) 为复选操作定义枚举变量

由于将处理 4 个不同的状态,因此为每个图像定义一个枚举会提高程序的可读性。

```

public:
    enum CheckState{ NOSTATE = 0, UNCHECKED, CHECKED, CHILD_CHECKED,
        SELF_AND_CHILD_CHECKED };
    enum CheckType{ CHECK, UNCHECK, TOGGLE, REFRESH };

```

(5) 设计 SetCheck 函数

当复选某个条目时,其父条目的状态也会被修改,以标识其子条目被复选。同样,当解除某条目的复选时,也需要检查是否需要修改其父条目的状态。另外,如果某个条目被移动位置,那么同样必须检查父条目状态是否需要修改。

取决于 nCheck 的不同取值,SetCheck 函数决定条目的新状态。如果 nCheck 为 REFRESH,则只检查其子条目是否应该被复选,接着函数再更新父条目的状态。清单 6-43 所示

为 SetCheck 函数的源代码,其中 hItem 指定了将被更新状态的条目,nCheck 指定了操作的类型,其取值为 CHECK、UNCHECK、TOGGLE 或 REFRESH。

清单 6-43 SetCheck()函数

```

BOOL CTreeCtrlEX::SetCheck( HTREEITEM hItem, CheckType nCheck )
{
    if( hItem == NULL )
        return FALSE;
    UINT nState = GetItemState( hItem, TVIS_STATEIMAGEMASK ) >> 12;
    // 确定新的复选状态
    if ( nCheck == CHECK )
    {
        if( nState == CHECKED || nState == SELF_AND_CHILD_CHECKED )
            return TRUE;
        switch( nState )
        {
            case UNCHECKED: nState = CHECKED; break;
            case CHILD_CHECKED: nState = SELF_AND_CHILD_CHECKED; break;
        }
    }
    else if( nCheck == UNCHECK )
    {
        if( nState == UNCHECKED || nState == CHILD_CHECKED )
            return TRUE;
        switch( nState )
        {
            case CHECKED: nState = UNCHECKED; break;
            case SELF_AND_CHILD_CHECKED: nState = CHILD_CHECKED; break;
        }
    }
    else if( nCheck == TOGGLE )
    {
        switch( nState )
        {
            case UNCHECKED: nState = CHECKED; break;
            case CHECKED: nState = UNCHECKED; break;
            case CHILD_CHECKED: nState = SELF_AND_CHILD_CHECKED; break;
            case SELF_AND_CHILD_CHECKED: nState = CHILD_CHECKED; break;
        }
    }
    else if( nCheck == REFRESH )
    {
        // 检查子条目以确定新状态
        BOOL bChildSelected = FALSE;
        HTREEITEM htiChild = GetChildItem( hItem );
        // 检查子条目
        while( htiChild )
        {
            UINT nChildState = GetItemState( htiChild, TVIS_STATEIMAGEMASK ) >> 12;

```

```

        if( nChildState != UNCHECKED && nChildState != NOSTATE )
        {
            bChildSelected = TRUE;
            break;
        }
        htiChild = GetNextItem( htiChild, TVGN_NEXT );
    }
    if( bChildSelected )
    {
        if( nState == CHECKED ) nState = SELF_AND_CHILD_CHECKED;
        else if( nState == UNCHECKED ) nState = CHILD_CHECKED;
    }
    else
    {
        if( nState == SELF_AND_CHILD_CHECKED ) nState = CHECKED;
        else if( nState == CHILD_CHECKED ) nState = UNCHECKED;
    }
}

SetItemState( hItem, INDEXTOSTATEIMAGEMASK(nState), TVIS_STATEIMAGEMASK );
if( nState == CHECKED || ( REFRESH && (nState == SELF_AND_CHILD_CHECKED
    || nState == CHILD_CHECKED)))
{
    // 标记父条目以标识其子条目被复选
    while( (hItem = GetParentItem( hItem )) != NULL )
    {
        nState = GetItemState( hItem, TVIS_STATEIMAGEMASK ) >> 12;
        if( nState == UNCHECKED )
            SetItemState( hItem, INDEXTOSTATEIMAGEMASK( CHILD_CHECKED ),
                TVIS_STATEIMAGEMASK );
        else if( nState == CHECKED )
            SetItemState( hItem,
                INDEXTOSTATEIMAGEMASK( SELF_AND_CHILD_CHECKED ),
                TVIS_STATEIMAGEMASK );
    }
}

else if( nState == UNCHECKED )
{
    // 如果其子条目都没有被复选,则父条目状态应该被修改
    while( (hItem = GetParentItem( hItem )) != NULL )
    {
        BOOL bChildSelected = FALSE;
        HTREEITEM htiChild = GetChildItem( hItem );
        // 检查子条目
        while( htiChild )
        {
            UINT nChildState = GetItemState( htiChild, TVIS_STATEIMAGEMASK ) >> 12;
            if( nChildState != UNCHECKED && nChildState != NOSTATE )
            {
                bChildSelected = TRUE;
            }
        }
    }
}

```



```

        break;
    }
    htiChild = GetNextItem( htiChild, TVGN_NEXT );
}
if( bChildSelected )
{
    // 由于还有其他被复选的子条目,父条目无需更新
    break;
}
else
{
    //由于没有其他被复选的子条目,父条目需要更新
    UINT nParentState = GetItemState( hItem, TVIS_STATEIMAGEMASK ) >> 12;
    if( nParentState == CHILD_CHECKED )
        SetItemState( hItem, INDEXTOSTATEIMAGEMASK(UNCHECKED),
            TVIS_STATEIMAGEMASK );
    else if( nParentState == SELF_AND_CHILD_CHECKED )
        SetItemState( hItem, INDEXTOSTATEIMAGEMASK(CHECKED),
            TVIS_STATEIMAGEMASK );
}
}
}
return TRUE;
}

```

(6) 向 OnLButtonDown 消息处理函数中添加代码,以标记复选框

当用户单击复选框时,复选框应该被标记。下面使用 HitTest 函数来确定鼠标是否单击了复选框。清单 6-44 所示为需要向 OnLButtonDown 函数中添加的代码:

清单 6-44 OnLButtonDown() 函数

```

void CTreeCtrlEX::OnLButtonDown(UINT nFlags, CPoint point)
{
    ...
    UINT uFlags = 0;
    HTREEITEM hItem = HitTest(point,&uFlags);
    if( uFlags & TVHT_ONITEMSTATEICON )
    {
        SetCheck( hItem, TOGGLE );
        return;
    }
    ...
}

```

(7) 向 OnKeyDown 消息处理函数中添加代码,以标记复选框

此步为用户操作提供了键盘支持,由于用户通常使用空格键进行复选,因此下面的代码也将处理空格键输入。具体的处理与第(6)步相似,只是此处不是用 HitTest 确定条目句柄,而是用 GetSelectedItem。清单 6-45 所示为 OnKeyDown 函数的源代码:

清单 6-45 OnKeyDown()函数

```

void CTreeCtrlEX::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    if( nChar == VK_SPACE )
    {
        HTREEITEM hItem = GetSelectedItem();
        SetCheck( hItem, TOGGLE );
        return;
    }
}

```

(8) 设计辅助函数

在类中还需要设计其他一些辅助函数用于封装常用的操作。例如,得到条目是否被复选(IsItemChecked),得到第一个被复选条目(GetFirstCheckedItem),得到下一个被复选条目(GetNextCheckedItem)以及得到上一个被复选条目等(GetPrevCheckedItem)。它们的源代码分别如清单 6-46、6-47、6-48 和 6-49 所示:

清单 6-46 IsItemChecked()函数

```

BOOL CTreeCtrlEX::IsItemChecked(HTREEITEM hItem)
{
    int iImage = GetItemState( hItem, TVIS_STATEIMAGEMASK ) >> 12;
    return iImage == CHECKED || iImage == SELF_AND_CHILD_CHECKED;
}

```

清单 6-47 GetFirstCheckedItem()函数

```

HTREEITEM CTreeCtrlEX::GetFirstCheckedItem()
{
    for ( HTREEITEM hItem = GetRootItem(); hItem != NULL; )
    {
        int iImage = GetItemState( hItem, TVIS_STATEIMAGEMASK ) >> 12;
        if( iImage == CHECKED || iImage == SELF_AND_CHILD_CHECKED )
            return hItem;

        if( iImage != CHILD_CHECKED )
        {
            HTREEITEM hti = GetNextItem( hItem, TVGN_NEXT );
            if( hti == NULL )
                hItem = GetNextItem( hItem );
            else
                hItem = hti;
        }
        else
            hItem = GetNextItem( hItem );
    }
    return NULL;
}

```

清单 6-48 GetNextCheckedItem() 函数

```

HTREEITEM CTreeCtrlEX::GetNextCheckedItem( HTREEITEM hItem )
{
    hItem = GetNextItem( hItem );
    while( hItem != NULL )
    {
        int iImage = GetItemState( hItem, TVIS_STATEIMAGEMASK ) >> 12;
        if ( iImage == CHECKED || iImage == SELF_AND_CHILD_CHECKED )
            return hItem;

        if( iImage != CHILD_CHECKED )
        {
            HTREEITEM hti = GetNextItem( hItem, TVGN_NEXT );
            if( hti == NULL )
                hItem = GetNextItem( hItem );
            else
                hItem = hti;
        }
        else
            hItem = GetNextItem( hItem );
    }
    return NULL;
}

```

清单 6-49 GetPrevCheckedItem() 函数

```

HTREEITEM CTreeCtrlEX::GetPrevCheckedItem( HTREEITEM hItem )
{
    for ( hItem = GetPrevItem( hItem ); hItem != NULL; hItem = GetPrevItem( hItem ) )
        if ( IsItemChecked( hItem ) )
            return hItem;
    return NULL;
}

```

6.9.3 改变条目的字体和颜色

默认情况下,树视图控件中的所有条目使用同样的字体和颜色。如果控件中不同条目使用不同的字体与颜色,就能够使控件显得更活泼,如图 6-3 所示。由于树视图控件并不支持设置单个条目的颜色和字体,我们只能在控件创建后实现不同的条目效果。换句话说,必须在控件绘制了窗口后,再使用定制代码绘制条目文本。

由于树视图控件将基于窗口字体计算条目的高度(所有条目的高度相同),因此控件只能使用较小的字体尺寸,以免条目文本相互重叠。此外,树视图控件会自动管理水平滚动条,所以需要考虑标签的宽度。

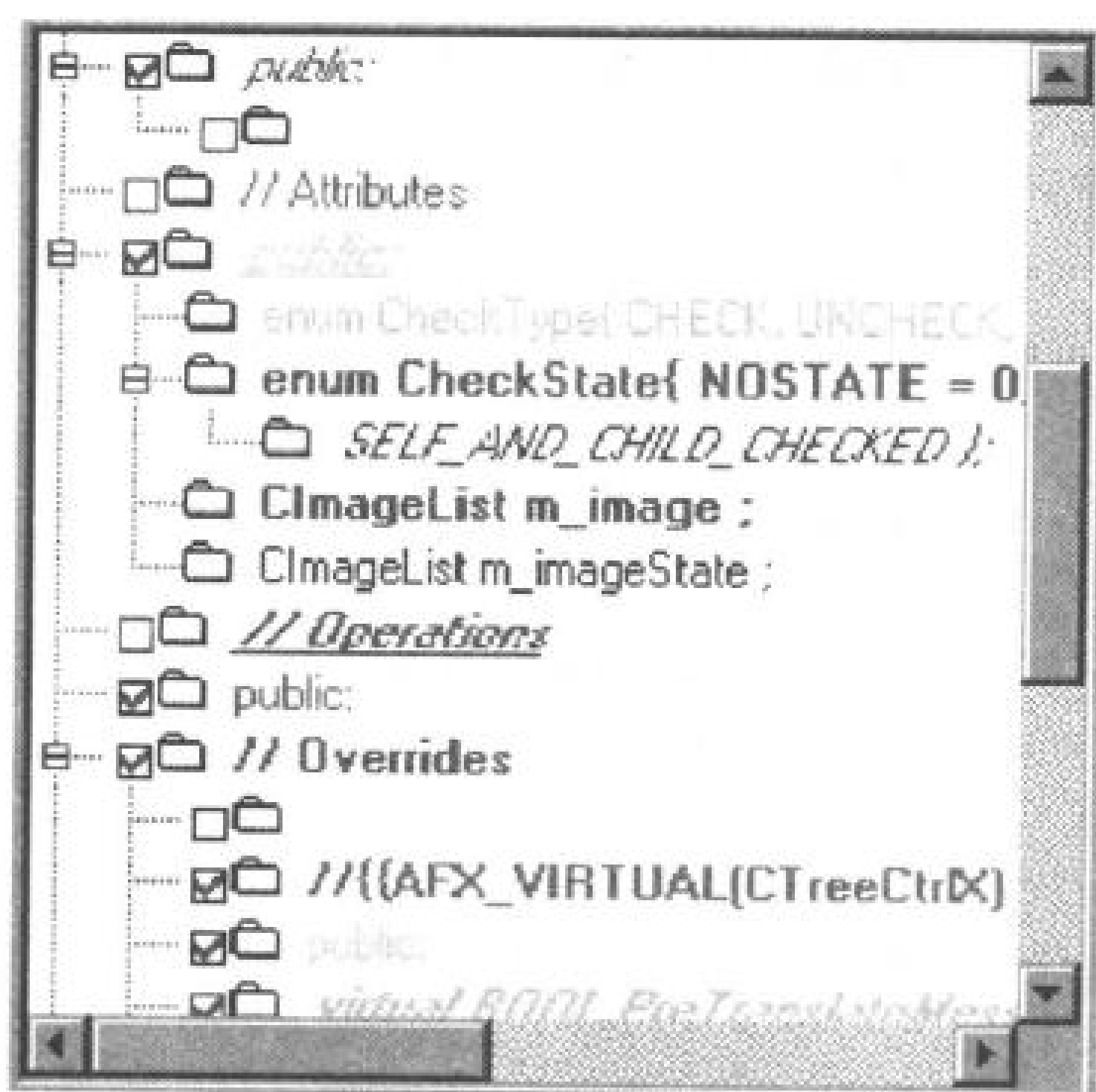


图 6-3 使用不同条目字体和颜色的树视图控件

(1) 添加成员变量以记录字体和颜色

控件不支持条目字体和颜色,因此必须在程序中自己记录。下面将使用 CMap 对象将字体和颜色性质映射到条目。映射中将只包含被显式改变的条目字体和颜色信息。在类定义中添加的声明如下:

```
protected:
    struct Color_Font
    {
        COLORREF color;
        LOGFONT logfont;
    };
    CMap< void*, void*, Color_Font, Color_Font& > m_mapColorFont ;
```

(2) 设计辅助函数以检索/设置字体和颜色

设置字体时将使用 LOGFONT 结构而不是字体句柄。下面定义的函数中有两个是用来检索和设置粗体属性。使用定制的函数而不是使用字体函数来处理粗体属性的原因主要有两个:首先,树视图控件支持将条目设置为粗体;其次,使用内建函数还能保持合适的水平滚动条设置。用于检索/设置文本属性的辅助函数分别如清单 6-50、6-51、6-52、6-53、6-54 和 6-55 所示:

清单 6-50 SetItemFont()函数

```
void CTreeCtrlEX::SetItemFont(HTREEITEM hItem, LOGFONT& logfont)
{
    Color_Font cf;
    if( ! m_mapColorFont.Lookup( hItem, cf ) )
        cf.color = (COLORREF)-1;
    cf.logfont = logfont;
    m_mapColorFont[hItem] = cf;
}
```

清单 6-51 SetItemBold() 函数

```
void CTreeCtrlEX::SetItemBold(HTREEITEM hItem, BOOL bBold)
{
    SetItemState( hItem, bBold ? TVIS_BOLD: 0, TVIS_BOLD );
}
```

清单 6-52 SetItemColor() 函数

```
void CTreeCtrlEX::SetItemColor(HTREEITEM hItem, COLORREF color)
{
    Color_Font cf;
    if( ! m_mapColorFont.Lookup( hItem, cf ) )
        cf.logfont.lfFaceName[0] = '\0';
    cf.color = color;
    m_mapColorFont[hItem] = cf;
}
```

清单 6-53 GetItemFont() 函数

```
BOOL CTreeCtrlEX::GetItemFont(HTREEITEM hItem, LOGFONT * plogfont)
{
    Color_Font cf;
    if( ! m_mapColorFont.Lookup( hItem, cf ) )
        return FALSE;
    if( cf.logfont.lfFaceName[0] == '\0' )
        return FALSE;
    *plogfont = cf.logfont;
    return TRUE;
}
```

清单 6-54 GetItemBold() 函数

```
BOOL CTreeCtrlEX::GetItemBold(HTREEITEM hItem)
{
    return GetItemState( hItem, TVIS_BOLD ) & TVIS_BOLD;
}
```

清单 6-55 GetItemColor() 函数

```
COLORREF CTreeCtrlEX::GetItemColor(HTREEITEM hItem)
{
    Color_Font cf;
    if( ! m_mapColorFont.Lookup( hItem, cf ) )
        return (COLORREF) - 1;
    return cf.color;
}
```

(3) 处理 WM_PAINT 消息

该消息处理函数首先使控件更新内存设备环境。接着将使用用户定义的属性重新绘

制可见标签,高亮条目则由控件处理,这样在更新条目标签前能够确定其不处于被选择状态。如果条目的字体或颜色属性没有改变,则无需进行定制绘制。当内存设备环境中的所有更新都已经完成时,就可以将其拷贝到实际设备环境中,这样控件就实现了彩色效果。为了避免出现闪烁,函数使用内存设备环境作为更新场所,而当更新完毕后才将其拷贝到主设备环境中。清单 6-56 所示为 OnPaint 函数的源代码:

清单 6-56 OnPaint()函数

```
void CTreeCtrlEX::OnPaint()
{
    CPaintDC dc(this);
    // 创建与绘制 DC 兼容的内存 DC
    CDC memDC;
    memDC.CreateCompatibleDC( &dc );

    CRect rcClip, rcClient;
    dc.GetClipBox( &rcClip );
    GetClientRect(&rcClient);

    // 将兼容位图选入内存 DC
    CBitmap bitmap;
    bitmap.CreateCompatibleBitmap( &dc, rcClient.Width(), rcClient.Height());
    memDC.SelectObject( &bitmap );

    // 将裁剪区域设置为与绘制 DC 相同
    CRgn rgn;
    rgn.CreateRectRgnIndirect( &rcClip );
    memDC.SelectClipRgn(&rgn);
    rgn.DeleteObject();

    // 首先使控件进行默认绘制
    CWnd::DefWindowProc( WM_PAINT, (WPARAM)memDC.m_hDC, 0 );
    HTREEITEM hItem = GetFirstVisibleItem();
    int n = GetVisibleCount() + 1;
    while( hItem && n-- )
    {
        CRect rect;
        UINT selflag = TVIS_DROPHILITED | TVIS_SELECTED;
        Color_Font cf;

        if ( ! (GetItemState( hItem, selflag ) & selflag )
            && m_mapColorFont.Lookup( hItem, cf ) )
        {
            CFont * pFontDC;
            CFont fontDC;
            LOGFONT logfont;
            if( cf.logfont.lfFaceName[0] != '\0' )
            {
                logfont = cf.logfont;
            }
            else
```

```

    {
        // 如果没有指定字体,则使用窗口字体
        CFont *pFont = GetFont();
        pFont->GetLogFont( &logfont );
    }
    if( GetItemBold( hItem ) )
        logfont.lfWeight = 700;
    fontDC.CreateFontIndirect( &logfont );
    pFontDC = memDC.SelectObject( &fontDC );
    if( cf.color != (COLORREF)-1 )
        memDC.SetTextColor( cf.color );
    CString sItem = GetItemText( hItem );
    GetItemRect( hItem, &rect, TRUE );
    memDC.SetBkColor( GetSysColor( COLOR_WINDOW ) );
    memDC.TextOut( rect.left+2, rect.top+1, sItem );

    memDC.SelectObject( pFontDC );
}
hItem = GetNextVisibleItem( hItem );
}
dc.BitBlt( rcClip.left, rcClip.top, rcClip.Width(), rcClip.Height(), &memDC,
           rcClip.left, rcClip.top, SRCCOPY );
}

```

在默认控件绘制完成后,上述代码检查所有可见条目,并更新那些用户定义了其字体和颜色属性的条目。

(4) 修改条目字体和颜色属性

示范代码如下:

```

// 改变条目颜色为红色
SetItemColor( hItem, RGB(255,0,0));

// 改变条目的字体并添加下划线
LOGFONT logfont;
CFont *pFont = GetFont();
pFont->GetLogFont( &logfont );
logfont.lfItalic = TRUE;
logfont.lfUnderline = TRUE;
SetItemFont(hti, logfont );

```

6.10 改善控件外观

6.9 节向读者介绍了改善条目的思路和方法,本节将介绍如何改善树视图控件本身的外观。

6.10.1 改变控件背景颜色

默认树视图控件背景为白色,本节将向读者介绍如何设置控件的背景颜色,如图6-4所示。与其他控件类似,树视图控件也可以处理 WM_CTLCOLOR 消息来改变背景色。不过当控件开始绘制条目时,它将忽略在 WM_CTLCOLOR 消息处理函数中设置的颜色而使用系统颜色。因此,这种方式是行不通的。

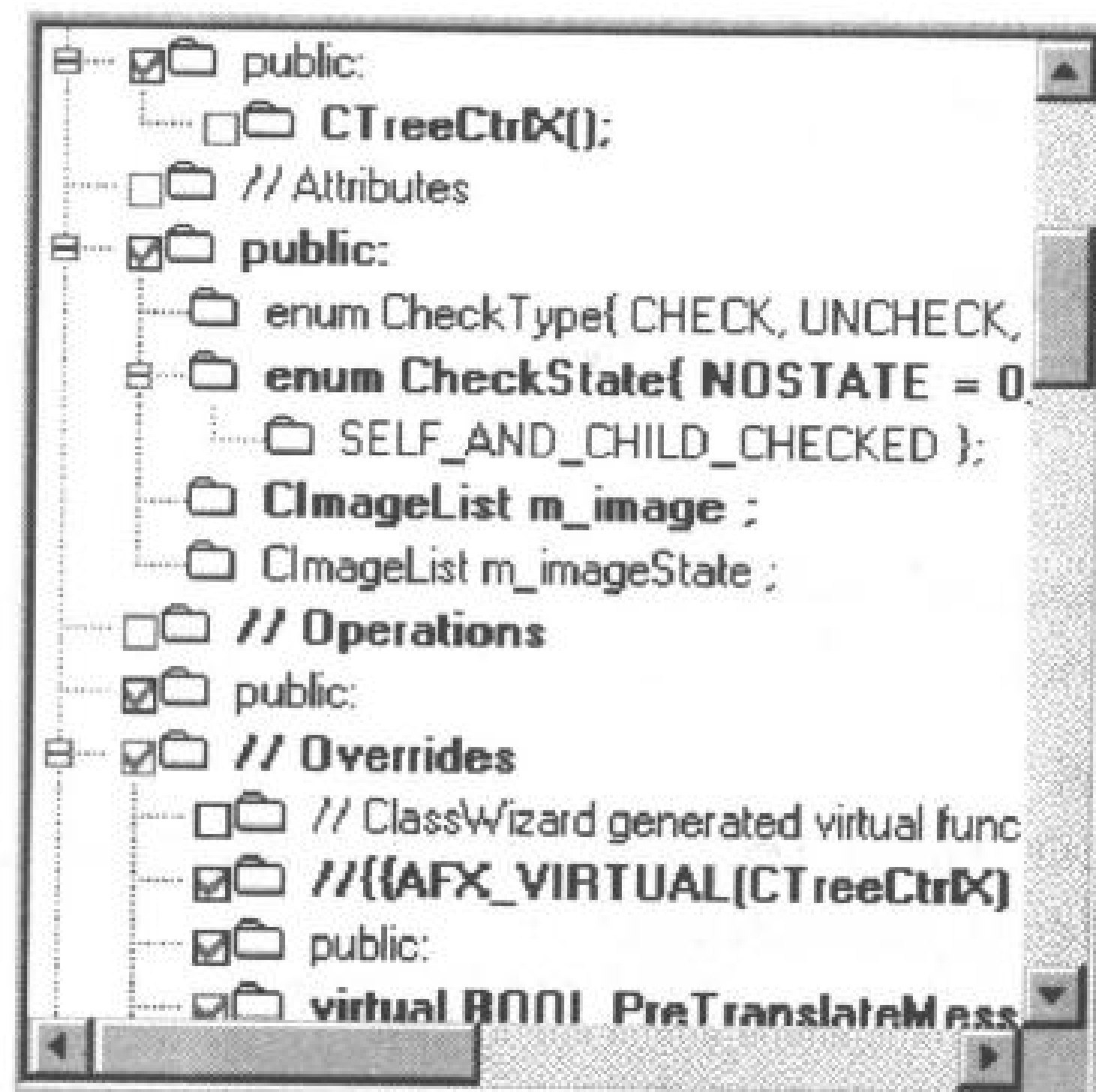


图 6-4 具有不同背景色的树视图控件

这里将使用如下的方法:当控件需要重绘时,使用均匀的颜色直接绘制背景。然后将绘制在内存设备环境中的条目图像,以透明的方式覆盖到控件的表面,从而达到改变背景色的目的。

(1) 添加 WM_PAINT 消息的处理函数

当控件中的内容被修改时,就会触发 WM_PAINT 消息,从而调用 OnPaint 函数进行重新绘制。函数首先创建与绘制 DC 兼容的内存 DC,然后将位图选入其中并进行合适的裁剪。接着使控件在内存 DC 上,以系统颜色作为背景色进行默认绘制。下一步就是使用另一个设备环境创建掩模位图。得到掩模位图后,使用绘制 DC 绘制背景色,然后将内存 DC 以透明方式绘制到绘制 DC 上。

清单 6-57 所示为 OnPaint 函数的源代码:

清单 6-57 OnPaint() 函数

```
void CTreeCtrlEX::OnPaint()
{
    CPaintDC dc(this);
    // 创建与绘制 DC 兼容的内存 DC
    CDC memDC;
    memDC.CreateCompatibleDC( &dc );

    CRect rcClip, rcClient;
```

```
dc.GetClipBox( &rcClip );
GetClientRect(&rcClient);

// 将兼容位图选入内存 DC 中
CBitmap bitmap;
bitmap.CreateCompatibleBitmap( &dc, rcClient.Width(), rcClient.Height());
memDC.SelectObject( &bitmap );

// 设置裁剪区域
CRgn rgn;
rgn.CreateRectRgnIndirect( &rcClip );
memDC.SelectClipRgn(&rgn);
rgn.DeleteObject();

// 首先使控件进行默认绘制
CWnd::DefWindowProc( WM_PAINT, (WPARAM)memDC.m_hDC, 0 );
// 创建掩模
CDC maskDC;
maskDC.CreateCompatibleDC(&dc);
CBitmap maskBitmap;

// 创建单色掩模位图
maskBitmap.CreateBitmap( rcClip.Width(), rcClip.Height(), 1, 1, NULL );
maskDC.SelectObject( &maskBitmap );
memDC.SetBkColor( ::GetSysColor( COLOR_WINDOW ));

// 创建用于内存 DC 的掩模
maskDC.BitBlt(0, 0, rcClip.Width(), rcClip.Height(), &memDC,
              rcClip.left, rcClip.top, SRCCOPY );

// 使用定制颜色填充背景
dc.FillRect(rcClip, &CBrush(RGB(255,255,192)));
// 将图像以透明方式拷贝到内存 DC 中
// 如使用 Windows NT 则下面调用 MaskBlt
//   dc.MaskBlt( rcClip.left, rcClip.top, rcClip.Width(), rcClip.Height(), &memDC,
//               rcClip.left, rcClip.top, maskBitmap, 0, 0,
//               MAKEROP4(SRCAND, SRCCOPY));

// 设置内存 DC 中的背景色为黑,使用黑色和其他颜色进行 SRCPAINT
// 不会改变其他颜色,因此使用黑色作为透明色
memDC.SetBkColor(RGB(0,0,0));
memDC.SetTextColor(RGB(255,255,255));
memDC.BitBlt( rcClip.left, rcClip.top, rcClip.Width(), rcClip.Height(),
              &maskDC, rcClip.left, rcClip.top, SRCAND);

// 将前景色设置为黑
dc.SetBkColor(RGB(255,255,255));
dc.SetTextColor(RGB(0,0,0));
dc.BitBlt( rcClip.left, rcClip.top, rcClip.Width(), rcClip.Height(),
           &maskDC, rcClip.left, rcClip.top, SRCAND);

// 组合前景色和背景色
dc.BitBlt(rcClip.left, rcClip.top, rcClip.Width(), rcClip.Height(), &memDC,
```



```
rcClip.left, rcClip.top, SRCPAINT);
```

(2) 添加 WM_ERASEBKGND 消息处理函数

由于已经在 OnPaint 函数中绘制了背景,因此只要使 WM_ERASEBKGND 消息处理函数返回 TRUE 即可。这确保不会调用默认的 Windows 进度擦除背景,从而阻止了对控件客户区附加的更新并减少了闪烁。不过在实际编程中,可能还需要先检查是否使用的是非默认背景色,如果是则返回 TRUE,否则返回 FALSE。清单 6-58 所示为 OnEraseBkgnd 函数的源代码:

清单 6-58 OnEraseBkgnd() 函数

```
BOOL CTreeCtrlEX::OnEraseBkgnd(CDC * pDC)
{
    return TRUE;
}
```

6.10.2 使用位图背景

使用位图作为树视图控件的背景能够大大改善控件的外观,如图 6-5 所示。在大部分情况下,这种方式要比改变控件的背景颜色效果要好,但是其实现也相对要复杂一些。由于树视图控件并不支持自绘制,这使得使用位图背景变得更加困难。实现这一点的基本想法就是先将控件内容绘制到内存 DC 中,然后将它再以透明方式绘制到背景图像上,然后将最终的图像拷贝到控件的客户区。

下面将向读者介绍如何使用 256 色位图(被添加为应用程序的位图资源)作为树视图控件的背景。在选择位图时,需要注意不要使位图影响对文本的阅读。如果位图比控件的尺寸小,它将在控件的客户区平铺显示。为了快速重绘,位图将与条目一起滚动。

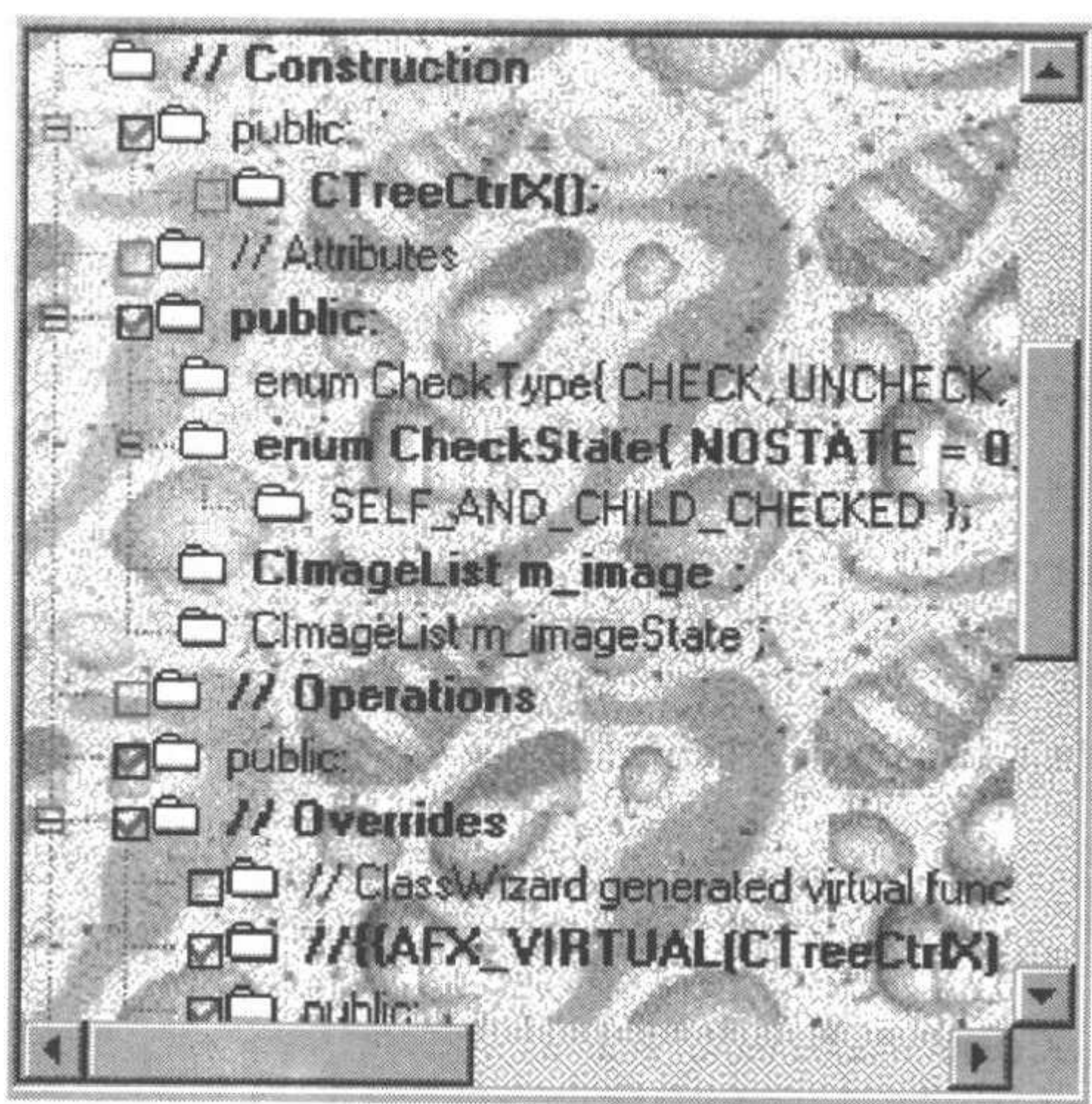


图 6-5 使用位图作为背景的树视图控件

(1) 添加位图资源。

(2) 在控件管理类定义中添加成员变量。显然,每当某个条目需要重绘时都重新载入位图或创建逻辑调色板是非常低效率的。因此,在类定义中为其添加成员变量以储存位图、位图尺寸和调色板:

```
protected:
    CPalette m_pal;
    CBitmap m_bitmap;
    int m_cxBitmap, m_cyBitmap;
```

(3) 添加设置背景位图的成员函数。

由于我们使用的位图是应用程序的资源,那么就有两种可能的调用方式:使用其 ID 或名称。在控件管理类中相应添加了两个不同版本的函数,其中之一将资源 ID 作为参数,而另一个则将资源名称作为参数。这两个函数执行的功能与 6.5.2 中相应的函数相同,源代码请读者参考清单 6-15。

(4) 添加 WM_PAINT 消息处理函数。

在 OnPaint 函数中将执行的操作与 6.10.1 节中设置背景色相似,实现方法在本节开头中已经介绍过了,这里就不再赘述。清单 6-59 所示为 OnPaint() 函数的源代码:

清单 6-59 OnPaint() 函数

```
void CTreeCtrlX::OnPaint()
{
    CPaintDC dc(this);
    CRect rcClip, rcClient;
    dc.GetClipBox( &rcClip );
    GetClientRect(&rcClient);

    // 创建兼容内存 DC
    CDC memDC;
    memDC.CreateCompatibleDC( &dc );

    // 将兼容位图选入内存 DC 中
    CBitmap bitmap, bmpImage;
    bitmap.CreateCompatibleBitmap( &dc, rcClient.Width(), rcClient.Height());
    memDC.SelectObject( &bitmap );

    // 首先进行默认绘制
    CWnd::DefWindowProc( WM_PAINT, (WPARAM)memDC.m_hDC, 0 );

    // 将位图绘制到背景中
    if( m_bitmap.m_hObject != NULL )
    {
        // 创建掩模
        CDC maskDC;
        maskDC.CreateCompatibleDC(&dc);
        CBitmap maskBitmap;

        // 创建单色位图掩模
        maskBitmap.CreateBitmap( rcClient.Width(), rcClient.Height(), 1, 1, NULL );
```

```

maskDC.SelectObject( &maskBitmap );
memDC.SetBkColor( ::GetSysColor( COLOR_WINDOW ) );

// 从内存 DC 创建掩模
maskDC.BitBlt(0, 0, rcClient.Width(), rcClient.Height(), &memDC,
               rcClient.left, rcClient.top, SRCCOPY );

CDC tempDC;
tempDC.CreateCompatibleDC(&dc);
tempDC.SelectObject( &m_bitmap );

CDC imageDC;
CBitmap bmpImage;
imageDC.CreateCompatibleDC( &dc );
bmpImage.CreateCompatibleBitmap( &dc, rcClient.Width(), rcClient.Height() );
imageDC.SelectObject( &bmpImage );

if( dc.GetDeviceCaps(RASTERCAPS) & RC_PALETTE && m_pal.m_hObject != NULL )
{
    dc.SelectPalette( &m_pal, FALSE );
    dc.RealizePalette();
    imageDC.SelectPalette( &m_pal, FALSE );
}

// 得到 x 和 y 偏移
CRect rcRoot;
GetItemRect( GetRootItem(), rcRoot, FALSE );
rcRoot.left = -GetScrollPos( SB_HORZ );

// 以平铺方式绘制位图
for(int i = rcRoot.left; i < rcClient.right; i += m_cxBitmap )
    for(int j = rcRoot.top; j < rcClient.bottom; j += m_cyBitmap )
        imageDC.BitBlt( i, j, m_cxBitmap, m_cyBitmap, &tempDC,
                        0, 0, SRCCOPY );

// 将内存 DC 背景设置为黑色
memDC.SetBkColor( RGB(0,0,0) );
memDC.SetTextColor( RGB(255,255,255) );
memDC.BitBlt( rcClip.left, rcClip.top, rcClip.Width(), rcClip.Height(), &maskDC,
              rcClip.left, rcClip.top, SRCAND );

// 设置前景色为黑色
imageDC.SetBkColor( RGB(255,255,255) );
imageDC.SetTextColor( RGB(0,0,0) );
imageDC.BitBlt( rcClip.left, rcClip.top, rcClip.Width(), rcClip.Height(), &maskDC,
                rcClip.left, rcClip.top, SRCAND );

// 组合前景与背景
imageDC.BitBlt( rcClip.left, rcClip.top, rcClip.Width(), rcClip.Height(),
                &memDC, rcClip.left, rcClip.top, SRCPAINT );
// 将最终的位图绘制到屏幕上
dc.BitBlt( rcClip.left, rcClip.top, rcClip.Width(), rcClip.Height(),
            &imageDC, rcClip.left, rcClip.top, SRCCOPY );

```

```

    }
    else
    {
        dc.BitBlt( rcClip.left, rcClip.top, rcClip.Width(),
                  rcClip.Height(), &memDC,
                  rcClip.left, rcClip.top, SRCCOPY );
    }
}

```

(5) 处理滚动消息。

处理滚动消息的唯一原因是因为这有助于减少因控件更新而引起的闪烁。对 WM_HSCROLL 和 WM_VSCROLL 消息的默认处理是：首先由 Windows 进程处理，然后重新绘制新显示的区域。通过调用 InvalidateRect 函数，能够确保控件只被更新一次。清单 6-60、6-61 所示为 OnVScroll 和 OnHScroll 函数的源代码：

清单 6-60 OnVScroll() 函数

```

void CTreeCtrlX::OnVScroll(UINT nSBCode, UINT nPos, CScrollBar * pScrollBar)
{
    if( m_bitmap.m_hObject != NULL )
        InvalidateRect(NULL);
    CTreeCtrl::OnVScroll(nSBCode, nPos, pScrollBar);
}

```

清单 6-61 OnHScroll() 函数

```

void CTreeCtrlX::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar * pScrollBar)
{
    if( m_bitmap.m_hObject != NULL )
        InvalidateRect(NULL);
    CTreeCtrl::OnHScroll(nSBCode, nPos, pScrollBar);
}

```

(6) 处理 TVN_ITEMEXPANDING 消息。

处理 TVN_ITEMEXPANDING 消息的原因与处理滚动消息的原因一样。清单 6-62 所示为 OnItemExpanding 函数的源代码：

清单 6-62 OnItemExpanding() 函数

```

void CTreeCtrlX::OnItemExpanding(NMHDR * pNMHDR, LRESULT * pResult)
{
    NM_TREEVIEW * pNMTreeView = (NM_TREEVIEW *)pNMHDR;
    if( m_bitmap.m_hObject != NULL )
        InvalidateRect(NULL);

    *pResult = 0;
}

```

(7) 处理 WM_ERASEBKGD 消息。

(8) 处理 WM_QUERYNEWPALETTE 和 WM_PALETTECHANGED 消息。

(9) 从顶层窗口转发调色板消息。

最后三步的处理与 5.5.2 中的最后三步处理相同,请读者参考 5.5.2 节中的介绍。

6.11 序列化树视图控件内容

序列化树视图控件中的内容能够用于保存对控件的改变,而更重要的则是能够保持控件上次被关闭时的状态。也就是说,上次关闭控件时其中被展开的节点,这次被打开时依然会保持展开;同理,上次被收拢的节点也依然会保持其收拢状态。

下面实现的 `Serialize` 函数能够将当前控件中的内容保存到一个文本文件,也能够从文本文件中读出控件的内容。在保存控件内容时, `tabs` 用于表示条目文本的缩进。而在读出控件内容时, `tabs` 则用于确定条目的级别,从而决定其放置位置。清单 6-63 所示为 `Serialize` 函数的源代码:

清单 6-63 `Serialize()` 函数

```
void CTreeCtrlEX::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        // 储存
        HTREEITEM hti = GetRootItem();
        while( hti )
        {
            int indent = GetIndentLevel( hti );
            while( indent-- )
                ar.WriteString( "\t" );
            ar.WriteString( GetItemText( hti ) + "\r\n" );
            hti = GetNextItem( hti );
        }
    }
    else
    {
        // 载入
        CString sLine;
        if( ! ar.ReadString( sLine ) )
            return;

        HTREEITEM hti = NULL;
        int indent, baseindent = 0;
        while( sLine[baseindent] == '\t' )
            baseindent ++;
        do
        {
            if( sLine.GetLength() == 0 )
                continue;
            for( indent = 0; sLine[indent] == '\t'; indent ++ );
```

```

sLine = sLine.Right( sLine.GetLength() - indent );
indent -= baseindent;

HTREEITEM parent;
int previndent = GetIndentLevel( hti );
if( indent == previndent )
    parent = GetParentItem( hti );
else if( indent > previndent )
    parent = hti;
else
{
    int nLevelsUp = previndent - indent;
    parent = GetParentItem( hti );
    while( nLevelsUp-- )
        parent = GetParentItem( parent );
}
hti = InsertItem( sLine, parent ? parent : TVI_ROOT, TVI_LAST );
while( ar.ReadString( sLine ) );
}
}

```

6.12 目录浏览器

树视图控件最常用于显示目录结构,本节向读者介绍如何创建目录浏览器,如图 6-6 所示。



图 6-6 目录浏览器

首先创建一个管理目录浏览树视图控件的类: CDirTreeCtrl。调用 Windows API 函数

SHGetSpecialFolderLocation 可以得到系统目录 ID 列表,然后调用 SHGetPathFromIDList 函数将得到的目录 ID 列表转换为文件系统路径。因此,只要在创建控件时(WM_CREATE)得到系统目录信息,并将其作为条目加入控件即可。实际上,只要解决了目录信息来源也就解决了目录浏览的问题。其他的处理并没有什么特别需要介绍的,请读者参考配套光盘 chap7/dirtree 目录下的源代码。

本章小结

本章主要向读者介绍了如何修改常规 Windows 树视图控件,通过本章学习读者应该达到以下几点:

- 掌握 CTreeCtrl 类的使用。
- 掌握定制树视图控件的方法。

第 7 章 菜 单

菜单是 Windows 应用程序的重要组成元素之一。大量的操作都是通过菜单命令完成的。虽然使用 AppWizard 生成的应用程序会自动创建标准框架菜单及其处理函数,但是为了满足实际需要,我们有时需要使用不同形式的菜单。本章将向读者介绍各种菜单类型的设计方法。

本章要点:

- CMenu 类的设计;
- 标准菜单的设计;
- 快捷菜单的设计;
- 动态菜单的设计;
- 自绘制菜单的设计。

7.1 菜单编程基础

菜单是由 CMenu 类管理的,图 7-1 显示了 CMenu 类的派生结构图。

可以在局部堆栈或全局堆中创建 CMenu 对象,然后调用 CMenu 类的成员函数对其进行操作。然后,调用 `CWnd::SetMenu` 将菜单挂接到某个窗口,接着立即调用 `CMenu::Detach` 函数。`CWnd::SetMenu` 函数将新菜单作为窗口菜单,并立即重新对其进行绘制。而 `Detach` 函数则解除 `HMENU` 与 `CMenu` 对象之间的联系,这样当局部的 `CMenu` 变量超出使用范围后,实际菜单也不会被销毁,只有当其父窗口被销毁时,它才自动被销毁。

使用 `LoadMenuIndirect` 成员函数能够从模板创建菜单,不过调用 `LoadMenu` 函数从菜单资源中创建菜单应该是更简单的方法,并且使用菜单编辑器还能够直接修改菜单。

使用 `CMenu` 类能够在应用程序中动态创建、跟踪、更新或销毁菜单,下面给出该类中常用的成员函数。

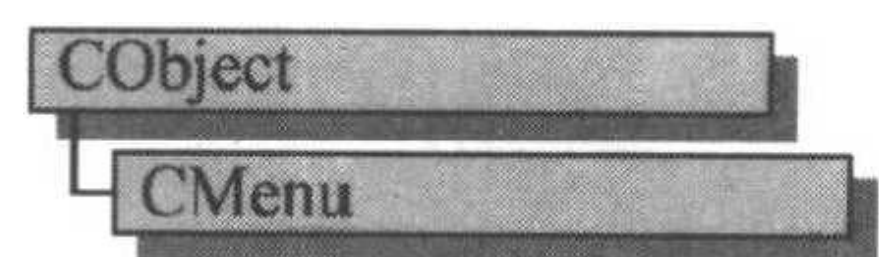


图 7-1 CMenu 类的派生结构图

7.1.1 构造函数

`CMenu` 的构造函数为: `CMenu`,调用该函数可以创建一个 `CMenu` 对象,其原型为:

```
CMenu( );
```

在创建了 CMenu 对象后,还必须调用 CreateMenu、CreatePopupMenu、LoadMenu、LoadMenuIndirect、或 Attach 函数以创建菜单。

7.1.2 初始化函数

CMenu 类的初始化函数包括: Attach、Detach、FromHandle、GetSafeHmenu、DeleteTempMap、CreateMenu、CreatePopupMenu、LoadMenu、LoadMenuIndirect 和 DestroyMenu,它们可以完成创建菜单、得到菜单句柄等操作。

- Attach

调用该函数将指定菜单句柄赋予一个 CMenu 对象,其原型如下:

```
BOOL Attach( HMENU hMenu );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

hMenu —— 指定了将与 CMenu 对象相联系的 Windows 菜单的句柄。

调用该函数可以将一个已经存在的菜单连接到 CMenu 对象上,菜单的句柄储存在 CMenu 对象的 m_hMenu 数据成员中。

- Detach

调用该函数将菜单句柄与 CMenu 对象之间断开联系,其原型如下:

```
HMENU Detach( );
```

返回值:

如果函数调用成功,则返回被断开连接的菜单句柄,否则返回 NULL。这时相应 CMenu 对象的 m_hMenu 数据成员被置为 NULL。

- FromHandle

调用该函数将获得一个指向与指定菜单句柄相联系的 CMenu 对象的指针,其原型如下:

```
static CMenu* PASCAL FromHandle( HMENU hMenu );
```

返回值:

如果函数调用成功,则返回 指向 CMenu 对象的指针。

其中参数 hMenu 为指定菜单的句柄。如果没有与该菜单相联系的 CMenu 对象,则函数将创建一个临时的 CMenu 对象,并将该对象与指定的菜单句柄相联系。

- GetSafeHmenu

调用该函数将获得 CMenu 对象的 m_hMenu 数据成员,其原型如下:

```
HMENU GetSafeHmenu( ) const;
```

返回值:

如果函数调用成功,则返回与 CMenu 对象相联系的 HMENU 句柄。如果该 CMenu 对

象尚未与菜单相联系,则返回 NULL 指针。

- DeleteTempMap

调用该函数将删除由 FromHandle 函数创建的任何临时 CMenu 对象,其原型如下:

```
static void PASCAL DeleteTempMap( );
```

该函数由 CWinApp 的空闲处理进程自动调用,在删除 CMenu 对象之前,函数会首先将菜单与对象断开连接。

- CreateMenu

调用该函数创建一个空菜单并将其赋予指定 CMenu 对象,其原型如下:

```
BOOL CreateMenu( );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

函数所创建的只是一个空的菜单对象,可以使用 AppendMenu 或 InsertMenu 成员函数向对象中加入菜单项。如果使用窗口指针调用函数,则该对象在窗口销毁时自动被销毁。如果菜单对象并不属于某个窗口的话,则在应用程序退出之前,必须释放所有与该菜单对象有关的系统资源,一般这种释放工作通过调用 DestroyMenu 成员函数完成。

- CreatePopupMenu

调用该函数创建一个弹出式空菜单并将其赋予指定的 CMenu 对象,其原型如下:

```
BOOL CreatePopupMenu( );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

函数所创建的只是一个空的弹出式菜单对象,可以使用 AppendMenu 或 InsertMenu 成员函数向对象中加入菜单项。使用该函数,应用程序可以将弹出式菜单项添加到现存的菜单上。使用 TrackPopupMenu 成员函数可以将弹出式菜单以浮动的方式显示在鼠标位置处。

同样如果使用窗口指针来调用函数,则该对象在窗口销毁时自动被销毁。而如果菜单对象并不属于某个窗口的话,则在应用程序退出之前,必须调用 DestroyMenu 成员函数释放所有与该菜单对象有关的系统资源。

- LoadMenu

调用该函数将指定菜单资源载入 CMenu 对象,其原型如下:

```
BOOL LoadMenu( LPCTSTR lpszResourceName );
```

```
BOOL LoadMenu( UINT nIDResource );
```

如果函数调用成功则返回非零值。其中参数 lpszResourceName 为包含菜单资源名称的字符串指针,参数 nIDResource 为菜单资源的 ID。该函数将指定的菜单资源载入并与 CMenu 对象相联系。在应用程序退出之前,必须调用 DestroyMenu 成员函数来释放所有与

该菜单对象有关的系统资源。

- LoadMenuIndirect

调用该函数将从内存菜单模板中载入菜单,并将其与 CMenu 对象相联系,其原型如下:

```
BOOL LoadMenuIndirect( const void* lpMenuTemplate );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

lpMenuTemplate —— 为指向菜单模板的指针。该模板是一个 MENUITEMTEMPLATEHEADER 结构和几个 MENUITEMTEMPLATE 结构的集合。

结构 MENUITEMTEMPLATEHEADER 定义了菜单模板的标头。完整的菜单模板包含标头和一个或几个菜单项列表,其定义如下:

```
typedef struct {
    WORD versionNumber;
    WORD offset;
} MENUITEMTEMPLATEHEADER;
```

结构成员:

versionNumber —— 指定了版本号,使用时必须设置为 0。

offset —— 指定了第一个 MENUITEMTEMPLATE 结构与标头的偏移量,由于菜单项的列表(即 MENUITEMTEMPLATE 结构的列表)紧跟着标头,因此通常该参数为 0。

MENUITEMTEMPLATE 结构定义了菜单中的一个菜单项,其定义如下:

```
typedef struct {
    WORD mtOption;
    WORD mtID;
    WCHAR mtString[1];
} MENUITEMTEMPLATE;
```

结构成员:

mtOption —— 指定了菜单项的标志,其取值可以为表 7-1 中某项或几项的联合。

表 7-1 mtOption 取值表

mtOption 取值	含义
MF_CHECKED	选择该标志,则菜单项旁边有一个复选框
MF_GRAYED	选择该标志,则菜单项初始状态为不激活态且变灰
MF_HELP	选择该标志,则菜单项的左边有一个垂直分隔
MF_MENUBARBREAK	选择该标志,则菜单项将置于新列中,新列和旧列之间以条分隔
MF_MENUBREAK	选择该标志,则菜单项将置于新列中
MF_OWNERDRAW	选择该标志,则菜单项将由主窗口绘制
MF_POPUP	选择该标志,则菜单项将为弹出式菜单

mtID —— 指定了命令 ID。

mtString —— 指定了菜单项字符串。

对于弹出式菜单中的最后一项或主菜单列表中的最后一项,必须将 mtOption 标志指定为 MF_END。如果为弹出式菜单项,则 mtId 为空。

- LoadMenu

调用该函数将销毁与指定 CMenu 对象相联系的菜单,并释放内存,其原型如下:

```
BOOL DestroyMenu( );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

7.1.3 菜单操作函数

CMenu 类的菜单操作函数包括: DeleteMenu 和 TrackPopupMenu,它们可以完成删除菜单、弹出浮动式菜单等操作。

- DeleteMenu

调用该函数将删除指定菜单的菜单项,其原型如下:

```
BOOL DeleteMenu( UINT nPosition, UINT nFlags );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

nPosition —— 指定了将要删除的菜单项。

nFlags —— 可以取 MF_BYCOMMAND 或 MF_BYPOSITION。MF_BYCOMMAND 为默认值,说明 nPosition 为命令 ID。如果 nFlags 取 MF_BYPOSITION,则 nPosition 为菜单项的位置。

如果被删除的菜单项还有子菜单,则 DeleteMenu 将同时销毁子菜单的句柄,并释放其所占用的内存。当窗口中的菜单发生变化时(无论窗口是否显示),应用程序必须调用 CWnd::DrawMenuBar 函数。

- TrackPopupMenu

调用该函数将在指定位置弹出浮动式菜单,其原型如下:

```
BOOL TrackPopupMenu( UINT nFlags, int x, int y, CWnd * pWnd, LPCRECT lpRect = NULL );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

nFlags —— 指定了屏幕位置和鼠标位置标志,该参数取值可以为表 7-2 中某项或几项的联合。

表 7-2 TrackPopupMenu 的 nFlags 取值表

nFlags 取值	含义
TPM_CENTERALIGN	快捷菜单相对于 x 居中
TPM_LEFTALIGN	快捷菜单相对于 x 左对齐
TPM_RIGHTALIGN	快捷菜单相对于 x 右对齐
TPM_LEFTBUTTON	单击鼠标左键出现快捷菜单
TPM_RIGHTBUTTON	单击鼠标右键出现快捷菜单

x —— 指定了弹出式菜单的左上角屏幕位置的 x 坐标。

y —— 指定了弹出式菜单的左上角屏幕位置的 y 坐标。

pWnd —— 指定了拥有弹出式菜单的窗口,该窗口将接收所有来自该菜单的 WM_COMMAND 消息。

lpRect —— 指定了鼠标可以单击且不取消弹出式菜单存在的矩形范围,如果该参数未被指定,则鼠标只要单击在弹出式菜单矩形之外,该菜单就会消失。

7.1.4 菜单项操作函数

CMenu 类的菜单项操作函数包括: AppnedMenu、CheckMenuItem、CheckMenuRadioItem、SetDefaultItem、GetDefaultItem、EnableMenuItem、GetMenuItemCount、GetMenuState、GetMenuItemInfo、GetSubMenu、InsertMenu、GetSubMenu、ModifyMenu、RemoveMenu、SetSubMenuBitmaps、GetMenuContextHelpId 和 SetMenuContextHelpId,它们可以完成对菜单项的操作。

• AppendMenu

调用该函数将在指定菜单末尾添加菜单项,其原型如下:

```
BOOL AppendMenu( UINT nFlags, UINT nIDNewItem = 0, LPCTSTR lpszNewItem = NULL );
BOOL AppendMenu( UINT nFlags, UINT nIDNewItem, const CBitmap* pBmp );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

nFlags —— 指定了新菜单项的状态信息,其取值可以为表 7-3 中某项或几项的联合。

表 7-3 AppendMenu 函数的 nFlags 取值表

nFlags 取值	含义
MF_CHECKED	所添加的菜单项将带有复选框,且已经被复选
MF_UNCHECKED	所添加的菜单项将带有复选框,但没有被复选
MF_DISABLED	所添加的菜单项将被禁止
MF_ENABLED	所添加的菜单项可以使用
MF_GRAYED	所添加的菜单项将被变灰,而不能使用
MF_MENUBARBREAK	所添加的菜单项将另起一行或一列,且与原来的菜单项之间由分隔线分开

续表

nFlags 取值	含义
MF_MENUBREAK	所添加的菜单项将另起一行或一列
MF_OWNERDRAW	所添加的菜单项将为主窗口绘制
MF_POPUP	所添加的菜单项为弹出式菜单
MF_SEPARATOR	所添加的菜单项为分隔
MF_STRING	所添加的菜单项为字符串

nIDNewItem —— 指定了菜单项的命令 ID。如果菜单项为弹出式菜单,则该参数为菜单句柄;如果菜单项为分隔线,则该参数为 NULL。

lpszNewItem —— 指定了新菜单项的内容。

如果 nFlags 取 MF_OWNERDRAW,则 lpszNewItem 包含了由应用程序提供的 32 位值,使用该值可以获得菜单项的附加数据。当应用程序处理 WM_MEASUREITEM 和 WM_DRAWITEM 消息时,该值可用。该值存储在这些消息所提供的结构的 itemData 成员中。参数 pBmp 指定了菜单项将使用的 CBitmap 对象。

• CheckMenuItem

调用该函数将在弹出式菜单项旁边添加或移去复选标志,其原型如下:

```
UINT CheckMenuItem( UINT nIDCheckItem, UINT nCheck );
```

返回值:

如果函数调用成功,则返回菜单项先前的状态: MF_CHECKED 或 MF_UNCHECKED。如果返回值为 0xFFFFFFFF 则表示该菜单项不存在。

参数:

nIDCheckItem —— 指定了将被检查的菜单项。

nCheck —— 指定了复选方式,其取值可以为表 7-4 中某项或几项的联合。

表 7-4 CheckMenuItem 函数的 nCheck 取值表

nFlags 取值	含义
MF_BYCOMMAND	nIDCheckItem, 该值为默认值
MF_BYPOSITION	nIDCheckItem 为菜单项的位置,如果是第一个菜单项,则 nIDCheckItem 取 0
MF_CHECKED	复选指定菜单项
MF_ENABLED	解除指定菜单项的选择

CheckMenuItem 函数的 nIDCheckItem 指定的是要被复选的菜单项,它既可以是常规菜单项,也可以是弹出式菜单项。这也就是说,不需要使用什么特殊的手段来复选弹出式菜单项。最顶端的菜单项是不能被复选的,例如 Word 中的“文件”菜单项。对于弹出式菜单,被复选的菜单项根据其位置确定,这是因为它们没有标识符(ID)。

如果 nFlag 为 MF_BYPOSITION,则应用程序必须确保使用的是正确的 CMenu 对象。如果使用的是与菜单栏联系的 CMenu 对象,则将影响顶部菜单项。要设置弹出式菜单项,或级联弹出式菜单项的状态,应用程序必须保证使用的是与弹出式菜单相联系的

CMenu 对象。

如果 nFlag 为 MF_BYCOMMAND, 则 Windows 检查 CMenu 对象拥有的所有的弹出式菜单项。因此, 除非存在完全相同的菜单项, 否则使用菜单栏的 CMenu 对象就完全可以了, 而不必向上面一样区分弹出式菜单和菜单栏菜单。

- CheckMenuItem

调用该函数将在指定菜单项之前放置单选按钮, 并且解除选项组中其他菜单项的单选按钮, 其原型如下:

```
BOOL CheckMenuItem( UINT nIDFirst, UINT nIDLast, UINT nIDItem, UINT nFlags );
```

返回值:

如果函数调用成功, 则返回非零值, 否则返回零值。

参数:

nIDFirst —— 指定了选项组中的第一个菜单项的 ID 或位置。

nIDLast —— 指定了选项组中的最后一个菜单项的 ID 或位置。

nIDItem —— 指定了将要被选择的菜单项的 ID 或位置。

nFlags —— 指定了前 3 个参数的取值规则。如果 nFlags 为 MF_BYCOMMAND, 说明取值规则为命令 ID。如果 nFlags 取 MF_BYPOSITION, 则取值规则为菜单项的位置。

- SetMenuItem

调用该函数将设置指定菜单默认菜单项, 其原型如下:

```
BOOL SetMenuItem( UINT uItem, BOOL fByPos = FALSE );
```

返回值:

如果函数调用成功, 则返回非零值, 否则返回零值。

参数:

uItem —— 指定了新的默认菜单项的 ID 或位置, 如果该参数为 -1, 则没有默认菜单项。

fByPos —— 指定了 uItem 的取值规则。如果该参数为真, 则 uItem 为菜单项位置, 否则 uItem 为命令 ID。

- GetMenuItem

调用该函数将获得指定菜单默认菜单项, 其原型如下:

```
UINT GetMenuItem( UINT gmdiFlags, BOOL fByPos = FALSE );
```

返回值:

如果函数调用成功, 则返回非零值, 否则返回零值。

参数:

gmdiFlags —— 指定了函数查找菜单项的方式。当 gmdiFlags 取 GMDI_GOINTOPOPUPS 时, 则如果没有默认菜单项, 函数将返回所搜寻菜单的标识符。如果 gmdiFlags 取 GMDI_USEDISABLED, 则即使默认菜单项被禁止, 函数也将返回该菜单项。

fByPos —— 指定了返回值的类型。如果该参数为真, 则返回菜单项位置, 否则返回

菜单项 ID。

- EnableMenuItem

调用该函数将使能、禁止或加灰指定菜单项,其原型如下:

```
UINT EnableMenuItem( UINT nIDEnableItem, UINT nEnable );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。函数将返回菜单项先前的状态: MF_DISABLED、MF_ENABLED 或 MF_GRAYED。如果返回值为 -1,则表示该菜单项不存在。

参数:

nIDEnableItem —— 指定了将要操作的菜单项。

nEnable —— 指定了将要采取的操作,它可以是以下几项之一或其联合: MF_DISABLED、MF_ENABLED、MF_GRAYED、MF_BYCOMMAND 或 MF_BYPOSITION。这几项的含义参见前表 4-3。

- GetMenuItemCount

调用该函数将获得指定弹出式菜单或顶端菜单中所包含的菜单项数,其原型如下:

```
UINT GetMenuItemCount( ) const;
```

返回值:

如果函数调用成功,则返回菜单项数,否则返回 -1。

- GetMenuItemID

调用该函数将获得指定位置处的菜单项 ID,其原型如下:

```
UINT GetMenuItemID( int nPos ) const;
```

返回值:

如果函数调用成功则返回菜单项 ID,否则返回 -1。

参数:

nPos —— 指定了要操作的菜单项的位置。

- GetMenuState

调用该函数将获得指定菜单项的状态,其原型如下:

```
UINT GetMenuState( UINT nID, UINT nFlags ) const;
```

返回值:

如果函数调用成功,则返回菜单项状态,否则返回 -1。

参数:

nID —— 指定了要操作的菜单项。

nFlags —— 指定了 nID 的取值规则。该参数可取值为: MF_BYCOMMAND 和 MF_BYPOSITION,其含义与前面介绍的相同。

- GetMenuString

调用该函数将获得指定菜单项的标题,其原型如下:

```
int GetMenuString( UINT nIDItem, LPTSTR lpString, int nMaxCount, UINT nFlags )
const;
int GetMenuString( UINT nIDItem, CString& rString, UINT nFlags ) const;
```

返回值:

如果函数调用成功,则返回所获得的字符串的长度。

参数:

nIDItem —— 指定了要操作的菜单项。

lpString —— 指定了将存储返回的字符串的缓冲区。

rString —— 指定了将返回字符串的 CString 对象。

nMaxCount —— 指定了缓冲区的最大长度,如果字符串的长度比缓冲区长,则超出的部分将被截去。

nFlags —— 指定了 nID 的取值规则。该参数可取值为: MF_BYCOMMAND 和 MF_BYPOSITION,其含义与前面介绍的相同。

• GetMenuItemInfo

调用该函数将获得指定菜单项的信息,其原型如下:

```
BOOL CMenu::GetMenuItemInfo( UINT nIDItem, LPMENUITEMINFO lpMenuItemInfo, BOOL
ByPos = FALSE );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

nIDItem —— 指定了要操作的菜单项。

lpMenuItemInfo —— 为 MENUITEMINFO 结构指针。

fByPos —— 指定了返回值的类型。如果该参数为真,则 nIDItem 为菜单项位置,否则为菜单项 ID。

其中结构 MENUITEMINFO 包含了菜单项的信息,其定义如下:

```
typedef struct tagMENUITEMINFO {
    UINT        cbSize;
    UINT        fMask;
    UINT        fType;
    UINT        fState;
    UINT        wID;
    HMENU       hSubMenu;
    HBITMAP     hbmpChecked;
    HBITMAP     hbmpUnchecked;
    DWORD       dwItemData;
    LPTSTR      dwTypeData;
    UINT        cch;
} MENUITEMINFO, FAR * LPMENUITEMINFO;
```

结构成员：
cbSize —— 指定了结构的尺寸。
fMask —— 指定了将获得或设置的信息，其取值如表 7-5 所示。

表 7-5 fMask 参数取值

fMask 取值	含义
MIIM_CHECKMARKS	将获取或设置 hbmpChecked 和 hbmpUnchecked 成员
MIIM_DATA	将获取或设置 dwItemData 成员
MIIM_ID	将获取或设置 wID 成员。
MIIM_STATE	将获取或设置 fState 成员
MIIM_SUBMENU	将获取或设置 hSubMenu 成员
MIIM_TYPE	将获取或设置 fType 和 dwTypeData 成员。

fType —— 指定了菜单项的种类，其取值如表 7-6 所示。

表 7-6 fType 参数取值

fType 取值	含义
MFT_BITMAP	使用位图显示菜单项，dwTypeData 成员的低位字为位图句柄，并忽略 cch 成员
MFT_MENUBARBREAK	将菜单项放置到一个新行(菜单栏)
MFT_MENUBREAK	将菜单项放置到一个新列(下拉式菜单、子菜单或快捷菜单)，此时将出现一个垂直分界线以区分新列和旧列
MFT_OWNERDRAW	拥有菜单的窗口将负责绘制菜单。菜单第一次显示之前，窗口会接收到 WM_MEASUREITEM 消息，而菜单项将被更新时，窗口会接收到 WM_DRAWITEM 消息。如果指定了该值，dwTypeData 成员为应用程序定义的 32 位值
MFT_RADIOCHECK	如果 hbmpChecked 成员为 NULL，则使用单选按钮标记表示菜单项被复选
MFT_RIGHTJUSTIFY	右对齐调整本菜单项及其后菜单项，该值只对菜单中的菜单项有效
MFT_RIGHTORDER	指定以从右到左的方式排列(默认为从左到右)，该值用于支持从右到左顺序的语言，例如阿拉伯语和希伯来语。该值用于 Windows 95、Windows NT 5.0 及以后的版本
MFT_SEPARATOR	指定菜单项为分隔符号，也就是菜单中常见的水平分隔线。此时忽略 dwTypeData 和 cch 成员。该值只对下拉式菜单、子菜单或快捷菜单有效
MFT_STRING	使用文本串显示菜单项，dwTypeData 为指向以空字符结尾的字符串，而 cch 为字符串的长度

MFT_BITMAP、MFT_SEPARATOR 和 MFT_STRING 不能相互组合使用。
fState —— 指定了菜单项的状态，其取值如表 7-7 所示。

表 7-7 fState 参数取值

fState 取值	含义
MFS_CHECKED	复选菜单项
MFS_DEFAULT	指定菜单项为默认菜单项,菜单中只能包括一个默认菜单项,它将以黑体显示
MFS_DISABLED	禁止菜单项使其不能被选择,但不将灰显
MFS_ENABLED	允许选择菜单项,这是默认取值
MFS_GRAYED	禁止菜单项使其不能被选择,并将其灰显
MFS_HILITE	高亮显示菜单项
MFS_UNCHECKED	解除菜单项的复选
MFS_UNHILITE	去除菜单项的高亮显示,这是默认取值

wID —— 指定了菜单项的 ID。
hSubMenu —— 指定了子菜单句柄。
hbmpChecked —— 指定了菜单项处于复选状态时的位图句柄。
hbmpUnChecked —— 指定了菜单项处于不复选状态时的位图句柄。
dwItemData —— 指定了与菜单项相联系的,由应用程序定义的值。
dwTypeData —— 指定了菜单内容。只有当 fMask 被设置为 MIIM_TYPE 时,该成员才会被使用。

在调用 GetMenuItemInfo 函数之前,应用程序必须为该成员分配一个缓冲区(其长度由 cch 成员给出)。如果所接收的菜单项为 MFT_STRING 类型,则 GetMenuItemInfo 将菜单项文本拷贝到缓冲区中。如果菜单项为其他类型,则 GetMenuItemInfo 将 dwTypeData 成员设置为由 fType 成员指定的类型的值。当使用 SetMenuInfo 函数时,该成员应该包含由 fType 指定的类型的值。

cch —— 如果菜单项为 MFT_STRING 类型,则该成员指定了菜单项文本长度。只有当 fMask 成员被设置 MIIM_TYPE 时,才使用该成员。在其他情况下,该成员为 0。当调用 SetMenuItemInfo 函数时,该成员被忽略。

在调用 GetMenuItemInfo 函数之前,应用程序必须将该成员设置为由 dwTypeData 指定的缓冲区的长度。如果所接收的菜单项为 MFT_STRING 类型,则 GetMenuItemInfo 将 cch 设置为字符串的长度。如果是其他类型的菜单项,则 GetMenuItemInfo 将 cch 设置为 0。

• GetSubMenu

调用该函数将获得指定弹出式菜单的指针,其原型如下:

```
CMenu * GetSubMenu( int nPos ) const;
```

返回值:

如果函数调用成功,则返回 CMenu 对象,否则返回 NULL。

参数:

nPos —— 指定了弹出式菜单的位置。

• InsertMenu

调用该函数将在指定位置插入一个新的菜单项,其原型如下:

```
BOOL InsertMenu( UINT nPosition, UINT nFlags, UINT nIDNewItem = 0, LPCTSTR lp-  
szNewItem = NULL );  
BOOL InsertMenu( UINT nPosition, UINT nFlags, UINT nIDNewItem, const CBitmap *  
pBmp );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

nPosition —— 指定了将在其前插入新菜单项的菜单项位置。

nFlags —— 指定了菜单排序规则,可取 MF_BYCOMMAND 和 MF_BYPOSITION。

nIDNewItem —— 指定了将插入的菜单项。如果 nFlags 被设置为 MF_POPUP,则该参数为菜单句柄(HMENU)。如果 nFlags 被设置为 MF_SEPARATOR,则 nIDNewItem 参数被忽略。

lpNewItem —— 指定了新菜单项的内容,其取值可为 MF_OWNERDRAW、MF_STRING 或 MF_SEPARATOR,含义与前面介绍的相同。

pBmp —— 指定了将用于菜单项的 CBitmap 对象指针。

当调用 InsertMenu 在指定位置插入菜单项时,该位置后的其他菜单项都将后移。应用程序通过设置 nFlags,能够指定菜单项的状态。当窗口中的菜单改变时,应用程序应该调用 CWnd::DrawMenuBar 进行绘制。

如果 nIDNewItem 指定的是一个弹出式菜单,则该菜单成为它将插入的主菜单的一部分。如果主菜单被销毁,则插入的菜单也同时被销毁。不过为了避免出现问题,最好在主菜单被销毁前,将插入菜单与 CMenu 对象解除连接。

在 MDI 应用程序中,当活动子窗口被最大化时,调用 InsertMenu 函数,并指定 MB_BYPOSITION 标志将一个弹出式菜单插入主菜单,则其插入位置会向左移一位。这是由于活动 MDI 子窗口的菜单是插入到 MDI 框架窗口菜单的第一个位置的。要使插入正确定位,应用程序必须将位置值加 1。应用程序可以使用 WM_MDIGETACTIVE 消息来确定当前子窗口是否被最大化。

- **ModifyMenu**

调用该函数修改指定位置处的菜单项,其原型如下:

```
BOOL ModifyMenu( UINT nPosition, UINT nFlags, UINT nIDNewItem = 0, LPCTSTR lp-  
szNewItem = NULL );  
BOOL ModifyMenu( UINT nPosition, UINT nFlags, UINT nIDNewItem, const CBitmap *  
pBmp );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

nPosition —— 指定了将要修改的菜单项。

nFlags —— 指定了菜单排序规则,可取 MF_BYCOMMAND 和 MF_BYPOSITION。

nIDNewItem —— 指定了修改后的菜单项。

lpstrNewItem —— 指定了新菜单项的内容,其取值可为 MF_OWNERDRAW、MF_STRING 或 MF_SEPARATOR,含义与前面介绍的相同。

pBmp —— 指定了将用于菜单项的 CBitmap 对象指针。

如果调用 ModifyMenu 函数,替换与菜单项相联系的弹出式菜单,则它首先销毁旧的弹出式菜单,并释放其所占用的内存。

- RemoveMenu

调用该函数删除指定的菜单项,其原型如下:

```
BOOL RemoveMenu( UINT nPosition, UINT nFlags );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

nPosition —— 指定了将要删除的菜单项。

nFlags —— 指定了菜单排序规则,可取 MF_BYCOMMAND 和 MF_BYPOSITION。

RemoveMenu 函数在删除某菜单项时,同时也将与其相关联的弹出式菜单删除。不过它并不销毁该弹出式菜单的句柄,这样该菜单可以被再次使用。在调用 RemoveMenu 函数之前,应用程序可能需要调用 GetSubMenu 成员函数,以得到与将被删除的菜单项相关联的弹出式菜单的 CMenu 对象,以便后用。

- SetMenuItemBitmaps

调用该函数将指定菜单项与位图对象相联系,其原型如下:

```
BOOL SetMenuItemBitmaps( UINT nPosition, UINT nFlags, const CBitmap * pBmpUnchecked, const CBitmap * pBmpChecked );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

nPosition —— 指定了将要操作的菜单项。

nFlags —— 指定了菜单排序规则,可取 MF_BYCOMMAND 和 MF_BYPOSITION。

pBmpUnchecked —— 指定了菜单项不处于复选状态的 CBitmap 对象。

pBmpChecked —— 指定了菜单项处于复选状态的 CBitmap 对象。

当菜单项被复选或取消选择时,窗口将在菜单项旁显示合适的位图。如果 pBmpUnchecked 或 pBmpChecked 为 NULL,则菜单项在相应状态下将不出现图案。如果两个参数都 NULL,则无论菜单项是否处于复选状态,都没有位图显示。当菜单被销毁时,这些位图并不被同时销毁,因此应用程序必须手动删除。

调用 GetMenuCheckMarkDimensions 函数,可以得到菜单项的默认复选标记尺寸。接着,应用程序可以使用这些值来确定位图的尺寸(根据尺寸创建或设置位图)。

- **GetMenuContextHelpId**

调用该函数获得指定菜单项的上下文帮助 ID,其原型如下:

```
DWORD GetMenuContextHelpId( ) const;
```

返回值:

如果函数调用成功,则返回上下文帮助 ID,否则返回 0。

- **SetMenuContextHelpId**

调用该函数设置指定菜单项的上下文帮助 ID,其原型如下:

```
BOOL SetMenuContextHelpId( DWORD dwContextHelpId );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

dwContextHelpId —— 指定了上下文帮助 ID。

调用这一函数将使所有菜单项与该上下文帮助 ID 相联系。目前还不能使单独的菜单项与上下文帮助 ID 相联系。

7.1.5 重载函数

CMenu 类的重载函数包括: DrawItem 和 MeasureItem,它们能够完成绘制菜单项和获得自绘制菜单项尺寸的操作。

- **DrawItem**

该函数由框架窗口调用,以绘制具有自绘制属性的菜单项,其原型如下:

```
virtual void DrawItem( LPDRAWITEMSTRUCT lpDrawItemStruct );
```

参数:

lpDrawItemStruct —— 为 DRAWITEMSTRUCT 结构。重载该函数可以按用户希望的方式绘制菜单项。

- **MeasureItem**

该函数由框架窗口调用,以确定绘制具有 OWNER-DRAW 属性的菜单项尺寸,其原型如下:

```
virtual void MeasureItem( LPMEASUREITEMSTRUCT lpMeasureItemStruct );
```

参数:

lpDrawItemStruct 为 DRAWITEMSTRUCT 结构。

当创建了一个具有自绘制属性的菜单项时,框架窗口调用该函数。默认情况下,该函数不作任何事。重载该函数并填充 DRAWITEMSTRUCT 结构,可以通知窗口要绘制的菜单项的尺寸。

7.2 使用标准菜单

使用 AppWizard 创建应用程序时, AppWizard 会为主窗口生成默认菜单资源, 当窗口被创建时, 该默认的菜单资源就会被自动载入。对于 MDI 应用程序, AppWizard 会自动生成 IDR_MAINFRAME 和 IDR_APPTYPE 菜单, 其中 APP 为应用程序名。用户可以像前几节所介绍的那样来定制默认菜单, 增添需要的菜单项或删除无用的菜单项。

使用这种默认菜单比较简单, 只需通过 ClassWizard 为相应菜单命令增加命令消息处理(WM_COMMAND)函数即可。

在应用程序运行中, 经常需要根据应用程序当前的状态对菜单项进行改变。例如对于 Edit|Copy 菜单项, 如果没有内容被选的话, 该菜单项就呈灰色被禁止使用。这就是说, 需要使应用状态与菜单项保持同步, 在 MFC 类库中采用了如下方法: 每当弹出式菜单第一次被显示时, 都会发送特殊的更新命令 UI 消息, 该消息通常被传递给与菜单项相联系的对象。更新命令 UI 消息的控制函数以一个 CCmdUI 对象作为其参数, 而该 CCmdUI 对象中包含了一个指向相应菜单项的指针, 于是控制函数就可以利用该指针对菜单项进行修改。更新命令 UI 消息只适用于弹出式菜单的菜单项和工具栏按钮, 对于始终显示的顶层菜单项则不适用。例如, 不能利用更新命令 UI 消息来禁止 Edit 菜单项。CCmdUI 类的常用成员函数如下所示:

- Enable

调用该函数允许或禁止用户接口对象, 如菜单、工具栏按钮等, 其原型为:

```
virtual void Enable( BOOL bOn = TRUE );
```

其中参数 bOn 为 TRUE 时, 将允许用户接口对象, 否则禁止用户接口对象。

- SetCheck

调用该函数设置用户接口对象的检查状态, 其原型为:

```
virtual void SetCheck( int nCheck = 1 );
```

其中参数 nCheck 为 1 时, 将复选该用户接口对象; 如果 nCheck 为 0 则取消该用户接口对象的复选状态; 如果 nCheck 为 2 则将该用户接口对象设置为不确定状态。

- SetRadio

调用该函数以设置单选组的选择状态, 其原型为:

```
virtual void SetRadio( BOOL bOn = TRUE );
```

其中参数 bOn 为 TRUE 时, 将选择该用户接口对象, 否则取消该用户接口对象的单选状态。与 SetCheck 函数不同, SetRadio 函数在单选某用户接口对象的同时取消单选组中其他对象的单选状态。

- SetText

调用该函数以设置用户接口对象的文本, 其原型为:

```
virtual void SetText( LPCTSTR lpszText );
```

其中参数 `lpszText` 为包含文本的字符串指针。

- **ContinueRouting**

调用该函数以通告消息循环机制继续运行,其原型为:

```
void ContinueRouting( );
```

这是一个高级函数,当 `ON_COMMAND_EX` 消息处理函数返回 `FALSE` 时,可以调用该函数继续消息循环,而不是退出或等待消息循环。

此外, `CCmdUI` 类的常用数据成员包括:

- `m_nID`: 该数据成员中存储了用户接口对象的 ID。
- `m_nIndex`: 该数据成员中存储了用户接口对象的索引。
- `m_pMenu`: 该数据成员为指向 `CCmdUI` 所代表的菜单的指针。如果该参数为 `NULL`,则表示该对象不是菜单。
- `m_pSubMenu`: 该数据成员为指向 `CCmdUI` 所代表的菜单中包含的第一个菜单项的指针。如果该参数为 `NULL`,则表示该对象不是菜单。
- `m_pOther`: 该数据成员为发送通告消息的 Windows 对象。

使用 `ClassWizard` 能够很容易地加入对更新命令 UI 消息的处理,只需在对象 ID 列表框中选择需要处理的菜单项 ID,并在消息列表框中选择 `UPDATE_COMMAND_UI` 消息,然后单击 `Add Function` 按钮即可。编程处理更新命令 UI 与处理一般消息命令的方法一样。

这里以显示/隐藏工具栏为例向读者介绍菜单命令消息处理函数的使用,程序如清单 7-1 所示。这类命令读者想必在许多 Windows 应用程序中都见到过,如果绘图工具栏没有显示在窗口中,则选择该命令将显示绘图工具栏,同时在该菜单命令左边出现一个复选标记;反之如果绘图工具栏显示在窗口中,则选择该命令将隐藏绘图工具栏,同时该菜单命令左边的复选标记消失。在程序中 `m_ToolBar` 代表着工具栏对象,程序首先调用其 `GetStyle` 成员函数得到该工具栏的状态。如果该工具栏可见, `bVisible` 的计算结果为 `FALSE`,即要隐藏绘图工具栏。在显示或隐藏工具栏时,调用了 `ShowControlBar` 函数,同时在该菜单命令的用户接口更新消息处理函数 `OnUpdateViewDrawbar` 中,调用 `SetCheck` 函数来设置菜单命令的复选状态。

清单 7-1 显示/隐藏工具栏菜单命令消息处理函数

```
void CMainFrame::OnViewDrawbar()
{
    //显示或隐藏绘图工具栏
    BOOL bVisible = ((m_ToolBar.GetStyle() & WS_VISIBLE) != 0);
    ShowControlBar(&m_ToolBar, ! bVisible, FALSE);
    RecalcLayout();
}

void CMainFrame::OnUpdateViewDrawbar(CCmdUI * pCmdUI)
{
    //显示或隐藏工具栏时相应设置菜单项状态(打勾或不打勾)
```

```

        BOOL bVisible = ((m_ToolBar.GetStyle() & WS_VISIBLE) != 0);
        pCmdUI->SetCheck(bVisible);
    }

```

7.3 使用快捷菜单

用户单击鼠标右键时弹出的菜单称为快捷菜单。快捷菜单为用户提供常用的命令列表可以加快操作速度。Visual C++ 专门为使用快捷菜单提供了 WM_CONTEXTMENU 消息。用户只要通过 ClassWizard 在应用程序中添加该消息的处理函数即可。示范代码如清单 7-2 所示,其中 IDR_CONTEXT_MENU 为快捷菜单资源。

清单 7-2 OnContextMenu() 函数

```

void CPainterView::OnContextMenu(CWnd* pWnd, CPoint point)
{
    // TODO: Add your message handler code here
    if( * DrawType == SELECT)
    {
        if (point.x == -1 && point.y == -1)
        {
            CRect rect;
            GetClientRect(rect);
            ClientToScreen(rect);
            point = rect.TopLeft();
            point.Offset(5, 5);
        }

        CMenu menu;
        VERIFY(menu.LoadMenu(IDR_CONTEXT_MENU));    //载入快捷菜单资源

        CMenu* pPopup = menu.GetSubMenu(0);
        ASSERT(pPopup != NULL);

        CWnd* pWndPopupOwner = this;    //设置快捷菜单所属的窗口
        while (pWndPopupOwner->GetStyle() & WS_CHILD)
            pWndPopupOwner = pWndPopupOwner->GetParent();

        pPopup->TrackPopupMenu(TPM_LEFTALIGN | TPM_RIGHTBUTTON, point.x,
            point.y,
            pWndPopupOwner);
    }
}

```

7.4 使用动态菜单

7.4.1 动态创建/修改菜单

应用程序可能需要动态的创建和改变菜单。这时用户可以定制另外的菜单资源,并

在应用程序中调用一些必要的函数,将菜单载入并激活。在应用程序中动态使用菜单一般要经过以下步骤:

- (1) 编辑菜单资源。
- (2) 为每个菜单项增加消息处理函数。
- (3) 将菜单载入应用程序中。

编辑菜单资源和为菜单项增加信息处理函数都很简单,通过资源编辑器和 ClassWizard 能够非常容易地完成,这在前面的内容中也都介绍过了,这里主要向读者介绍如何将菜单载入应用程序中。

首先创建 CMenu 对象,然后调用该对象的成员函数来对菜单进行设置和操作。设置完毕后,调用 CWnd::SetMenu 函数将菜单与窗口相联系,此时必须注意马上调用 CMenu 对象的 Detach 成员函数来解除菜单与 CMenu 对象的联系。调用 CWnd::SetMenu 后,新菜单将取代原有的菜单。调用 Detach 函数的目的在于,如果 CMenu 变量超出了使用范围不会导致菜单的销毁。该菜单将在窗口销毁时自动被销毁。用户可以使用 LoadMenuIndirect 成员函数从菜单模板中创建菜单,但是调用 LoadMenu 函数将资源中的菜单载入应用程序则更加方便些。实际上对于菜单资源的修改和编辑,使用菜单资源编辑器更加方便。因此,如非必要,一般不要使用 ModifyMenu 等成员函数在程序中动态修改菜单。

如果用户需要在 MDI 应用程序的不同子框架窗口中使用不同的菜单栏,那么一个简单的方法就是在窗口类的定义和注册时(即重载 Create 函数中),把菜单资源名称赋给窗口类的 m_hMenuShared 成员。m_hMenuShared 成员表示窗口类所使用的菜单资源,这样在该窗口创建时,菜单资源也会被载入。这比使用 SetMenu 要更方便些。m_hMenuShared 成员存储了窗口所拥有的菜单句柄。下面是典型的实现代码:

```
CMenu m_Menu;    //声明菜单对象
m_Menu.LoadMenu(IDR_WEBBROTYPE);    //将菜单资源连接到该菜单对象
m_hMenuShared = m_Menu.m_hMenu;    //将该菜单对象值赋予 m_hMenuShared
```

还有一种情况,用户需要动态地对应用程序菜单进行增删,例如在某种情况下需要将某菜单项中的菜单命令增加或减少。这时为这几种状态分别定制菜单资源,未免小题大做了,而且也是对系统资源的浪费。对于这种情况,一般先调用 CreateMenu 函数创建一个空的菜单栏,然后调用 CreatePopupMenu 函数创建一个空的菜单。在下面的示范代码中,为了在主菜单中加入一个弹出式菜单,首先调用 GetMenu 函数得到主菜单的句柄,然后调用 CreatePopupMenu 函数创建一个空的弹出式菜单,再调用 InsertMenu 将空的弹出式菜单加入到主菜单的指定位置处,最后使用 AppendMenu 函数向该弹出式菜单中添加菜单项。示范代码如下:

```
CMenu * MainMenu;    //主菜单对象
CMenu * SubMenu;    //要新建的菜单对象

MainMenu = AfxGetMainWnd()->GetMenu();    //将主框架窗口的菜单赋予主菜单对象
SubMenu.CreatePopupMenu();    //创建一个新的弹出式菜单

//在主菜单中的第 5 个位置插入新的弹出式菜单
```



```

MainMenu.InsertMenu((UINT)4, MF_STRING|MF_POPUP, (UINT)SubMenu, "新菜单");
//在弹出式菜单中添加菜单项
SubMenu.AppendMenu(MF_STRING, Doc -> m_aColorDef[i].m_nID, "菜单项 1");
SubMenu.AppendMenu(MF_STRING, Doc -> m_aColorDef[i].m_nID, "菜单项 2");

```

7.5 使用自绘制菜单

仅仅使用 Windows 直接提供的几种菜单项显示方式,例如文本、文本加复选框或文本加图标等,有时显得比较单调。如果使用更灵活的自绘制方式,可能会使你的应用程序增光不少。

但是使用自绘制菜单需要用户进行较多的编程工作,一般需要重载 CMenu 类的 DrawItem 函数,以添加所需的菜单项绘制代码。

7.5.1 彩色菜单

下面将创建一个由 CMenu 类派生的菜单类 CColorMenu,在应用程序中使用该类可以创建彩色菜单项。该类的定义代码如清单 7-3 所示:

清单 7-3 CColorMenu 类的类定义

```

#define COLOR_BOX_WIDTH 20
#define COLOR_BOX_HEIGHT 20

class CColorMenu : public CMenu
{
public:
// Operations
    void AppendColorMenuItem(UINT nID, COLORREF color);
// Implementation
    virtual void MeasureItem(LPMEASUREITEMSTRUCT lpMIS);
    virtual void DrawItem(LPDRAWITEMSTRUCT lpDIS);
    CColorMenu();
    virtual ~CColorMenu();
};

```

其中 COLOR_BOX_WIDTH 和 COLOR_BOX_HEIGHT 为颜色菜单项的宽度和高度。类的构造和析构函数如清单 7-4、7-5 所示,在构造函数中调用 CreateMenu 函数为 CColorMenu 类对象创建了一个菜单资源,在类的析构函数中将创建的菜单资源与 CColorMenu 类对象断开连接并将其删除(令类的 m_hMenu 成员为空,该成员的定义在基类 CMenu 中)。在构造和析构函数中使用的 VERIFY 和 ASSERT 为诊断宏,这些宏用于判断其条件是否合法。

清单 7-4 CColorMenu 类的构造函数

```

CColorMenu::CColorMenu()
{
    VERIFY(CreateMenu());
}

```

清单 7-5 CColorMenu 类的析构函数

```

CColorMenu::~CColorMenu()
{
    Detach();
    ASSERT(m_hMenu == NULL);    // default CMenu::~CMenu will destroy
}

```

AppendColorMenuItem 函数实际只包含一条语句,如清单 7-6 所示。该函数调用 AppendMenu 成员函数为与 CColorMenu 类对象相联系的菜单项添加菜单命令。定义该函数的主要目的是为了添加菜单命令时比较方便。注意这里在调用 AppendMenu 时,将用于定义菜单命令颜色的 COLORREF 型变量,强制转换成为 LPCTSTR 型变量作为函数的第四个参数使用。重载的 DrawItem 函数将该变量类型再转换为 COLORREF 使用。

清单 7-6 AppendColorMenuItem() 函数

```

void CColorMenu::AppendColorMenuItem(UINT nID, COLORREF color)
{
    VERIFY(AppendMenu(MF_ENABLED | MF_OWNERDRAW, nID, (LPCTSTR)color));
}

```

由于我们定义的 CColorMenu 类为一个自绘制菜单类,因此需要重载 MeasureItem 函数以填充 DRAWITEMSTRUCT 结构,从而通知窗口要绘制的菜单命令的尺寸,即 COLOR_BOX_WIDTH 和 COLOR_BOX_HEIGHT。该函数源代码如清单 7-7 所示:

清单 7-7 MeasureItem() 函数

```

void CColorMenu::MeasureItem(LPMEASUREITEMSTRUCT lpMIS)
{
    lpMIS->itemWidth = COLOR_BOX_WIDTH;
    lpMIS->itemHeight = COLOR_BOX_HEIGHT;
}

```

重载的 DrawItem 函数实现了在与 CColorMenu 对象相联系的菜单项中绘制颜色条菜单命令,源代码如清单 7-8 所示。菜单命令的颜色储存在 LPDRAWITEMSTRUCT 结构的 itemData 中。在前面 CMenu 类的介绍中,读者也许还记得,在使用 AppendMenu 函数时,如果指定了 MF_OWNERDRAW 标志,那么该函数的第 4 个参数将被作为附加数据。该附加数据通过 WM_MEASUREITEM 或 WM_DRAWITEM 消息处理函数(MeasureItem 函数或 DrawItem 函数)所带的 LPDRAWITEMSTRUCT 结构参数 lpDIS 的 itemData 数据成员来获得。该参数的 rcItem 成员数据在重载的 MeasureItem 函数中指定。

清单 7-8 DrawItem() 函数

```

void CColorMenu::DrawItem(LPDRAWITEMSTRUCT lpDIS)

```

```

{
    CDC * pDC = CDC::FromHandle(lpDIS->hDC);
    COLORREF cr = (COLORREF)lpDIS->itemData;    // 菜单项数据中的 RGB 颜色

    if (lpDIS->itemAction & ODA_DRAWENTIRE)
    {
        // 以指定的颜色绘制颜色条
        CBrush br(cr);
        pDC->FillRect(&lpDIS->rcItem, &br);
    }

    if ((lpDIS->itemState & ODS_SELECTED) &&
        (lpDIS->itemAction & (ODA_SELECT | ODA_DRAWENTIRE)))
    {
        // 当菜单命令被选择时,在颜色条周围绘制边框
        COLORREF crHilite = RGB(255-GetRValue(cr),
                                255-GetGValue(cr), 255-GetBValue(cr));
        CBrush br(crHilite);
        pDC->FrameRect(&lpDIS->rcItem, &br);
    }

    if (! (lpDIS->itemState & ODS_SELECTED) &&
        (lpDIS->itemAction & ODA_SELECT))
    {
        // 当菜单命令不处于选择状态时,去除环绕的边框
        CBrush br(cr);
        pDC->FrameRect(&lpDIS->rcItem, &br);
    }
}
}

```

下面我们新建了一个 ColorMenu 的 MDI 应用程序,在工程设置的各个步骤中都选择默认选项。在实际使用时,可通过调用 AttachCustomMenu 函数来向应用程序菜单中插入自绘制颜色条菜单。其中 ColorMenu 为应用程序的框架窗口管理类,常量 IDS_COLOR_NAME_FIRST 为自定义的字符串常量,该常量包含颜色条菜单中的第一个菜单命令的文本描述。由于菜单中的菜单命令是自绘制的颜色条,而不是使用菜单资源编辑器来创建的。因此为了方便指定其命令 ID,在应用程序中定义了 IDS_COLOR_NAME_FIRST、IDS_COLOR_NAME_BLUE 等一系列字符串常量,它们被用于作为自绘制菜单命令的 ID。示范代码如清单 7-9 所示,其中 m_colorMenu 为 CColorMenu 类对象。

清单 7-9 AttachCustomMenu() 函数

```

static COLORREF colors[] = {
    0x00000000,    // 黑色
    0x00FF0000,    // 蓝色
    0x0000FF00,    // 绿色
    0x00FFFF00,    // 青色
    0x000000FF,    // 红色
    0x00FF00FF,    // 洋红色
    0x0000FFFF,    // 黄色
    0x00FFFFFF     // 红色
}

```

```

};
const int nColors = sizeof(colors)/sizeof(colors[0]);

void ColorMenu::AttachCustomMenu()
{
    for (int iColor = 0; iColor < nColors; iColor++)
        m_colorMenu.AppendColorMenuItem ( IDS_COLOR_NAME_FIRST + iColor,
            colors[iColor]);

    // 将指定的菜单项改变为自绘制菜单
    CMenu * pMenuBar = GetMenu();
    ASSERT(pMenuBar != NULL);
    TCHAR szString[256];

    // IDM_TEST_CUSTOM_MENU 为在菜单编辑器中编辑的菜单项
    // 该菜单项是自绘制颜色条菜单的替身
    pMenuBar->GetMenuString(IDM_TEST_CUSTOM_MENU, szString, sizeof
        (szString),
        MF_BYCOMMAND);
    VERIFY(GetMenu()->ModifyMenu(IDM_TEST_CUSTOM_MENU, MF_BYCOMMAND |
        MF_POPUP, (UINT)m_colorMenu.m_hMenu, szString));
}

```

图 7-2 为程序运行时显示的颜色菜单的情况。

此时还存在一个问题：虽然彩色菜单被绘制出来了，但是当用户选择了其中的菜单命令时，框架应该如何处理？这一般需要调用 `OnCommand` 函数，它将参照 `ON_COMMAND` 消息映射入口来选择合适的消息处理函数。

重载 `OnCommand` 函数可以加入更灵活的处理方式，如果在重载的函数中不调用基类的 `OnCommand` 函数，那么重载函数不会执行消息映射。该函数的原型为：



图 7-2 彩色菜单

```
virtual BOOL OnCommand( WPARAM wParam, LPARAM lParam );
```

如果函数对消息进行了处理，则返回非零值，否则返回零值。其中参数 `wParam` 的低位字为对象的命令 ID，而如果发送消息的对象为控件，则高位字包含了通告消息的来源标志。如果消息来自加速键，则高位字为 1。如果消息来自菜单命令，则高位字为 0。如果发送消息的是控件，那么参数 `lParam` 标识了该控件，否则该参数为 0。

在上一节中创建的自绘制菜单的命令 ID（也代表着一系列字符串常量）值之间依次只相差 1。也就是说，这些命令 ID 是连续的，而且对这些菜单命令的处理方式也相似（在屏幕上显示一个提示消息，告诉用户选择的是什么颜色）。因此不必要分别为这些菜单命令创建消息处理函数，清单 7-10 的示范代码是重载的 `OnCommand` 函数，在该函数中对彩色菜单的各个菜单命令进行了处理。

清单 7-10 重载的 `OnCommand()` 函数

```

BOOL ColorMenu::OnCommand(WPARAM wParam, LPARAM lParam)
{

```

```

    if (wParam < IDS_COLOR_NAME_FIRST || wParam >= IDS_COLOR_NAME_FIRST +
        nColors)
        return CFrameWnd::OnCommand(wParam, lParam);

    // 选择特殊颜色
    CString strYouPicked;
    strYouPicked.LoadString(IDS_YOU_PICKED_COLOR);

    CString strColor;
    strColor.LoadString(wParam);

    CString strMsg;
    strMsg.Format(strYouPicked, (LPCTSTR)strColor);

    CString strMenuTest;
    strMenuTest.LoadString(IDS_MENU_TEST);

    MessageBox(strMsg, strMenuTest);

    return TRUE;
}

```

函数首先使用一个 if 语句判断发送命令消息的对象 ID 是否是颜色菜单命令 ID。读者可以发现,由于颜色菜单命令 ID 是连续的,因此使得判断操作也十分简单。另外,由于颜色菜单命令 ID 同时也是一个字符串常量 ID,因此可以将该 ID 传递给 CString 对象的 Load 成员函数,以初始化屏幕提示信息。

7.5.2 图标菜单

在各种软件的菜单中,经常可以发现菜单命令左边显示的图标。这些图标能够直观地说明菜单命令的用途。使用自绘制菜单也能很容易地实现这一点:只需将菜单项文本和图标同时绘制出来即可。首先创建图标菜单的管理类:TCMenu,其定义如清单 7-11 所示:

清单 7-11 TCMenu 类的定义

```

class TCMenu : public CMenu
{
public:
    TCMenu();

protected:
    CTypedPtrArray<CPtrArray, TCMenuData*> m_MenuList;    // 存储菜单项列表
    CTypedPtrArray<CPtrArray, TCMenu*> m_SubMenus;        // 存储子菜单列表

public:
    virtual ~TCMenu();
    virtual void DrawItem( LPDRAWITEMSTRUCT);    // 绘制菜单项
    virtual void MeasureItem( LPMEASUREITEMSTRUCT );    // 测量菜单项

    void SetTextColor( COLORREF );    // 设置文本颜色
    void SetBackColor( COLORREF );    // 设置背景颜色
}

```



```

void SetHighlightColor (COLORREF);    // 设置高亮颜色
void SetIconSize (int, int);    // 设置图标尺寸
void SetHighlightStyle (HIGHLIGHTSTYLE);    // 设置高亮风格
void SetHighlightTextColor (COLORREF);    // 设置高亮文本颜色

// 添加菜单项
virtual BOOL AppendODMenu(LPCTSTR lpstrText, UINT nFlags = MF_OWNERDRAW,
UINT nID = 0,UINT nIconNormal = -1, UINT nIconSelected = -1, UINT nIconDis-
abled = -1);
virtual BOOL ModifyODMenu(LPCTSTR lpstrText,UINT nID = 0,UINT nIconNormal = -1,
UINT nIconSelected = -1,UINT nIconDisabled = -1);    // 修改菜单项
virtual BOOL LoadMenu(LPCTSTR lpszResourceName);    // 载入菜单
virtual BOOL LoadMenu(int nResource);
virtual BOOL DestroyMenu();

protected:
COLORREF m_crText;
COLORREF m_clrBack;
COLORREF m_clrText;
COLORREF m_clrHilight;
COLORREF m_clrHilightText;
LOGFONT m_lf;
CFont m_fontMenu;
UINT m_iMenuHeight;
BOOL m_bLBtnDown;
CBrush m_brBackground,m_brSelect;
CPen m_penBack;
int m_iconX,m_iconY;
HIGHLIGHTSTYLE m_hilightStyle;
};

```

在类中定义的成员变量主要用于存储菜单颜色、字体、状态等信息;而成员函数则主要用于完成绘制菜单、插入菜单项、修改菜单项等操作。读者现在应该很清楚,自绘制菜单的实现主要是通过 DrawItem 函数绘制完成的。清单 7-12 所示为 DrawItem 函数的源代码:

清单 7-12 DrawItem() 函数

```

void TCMenu::DrawItem (LPDRAWITEMSTRUCT lpDIS)
{
    ASSERT(lpDIS != NULL);

    CDC * pDC = CDC::FromHandle(lpDIS->hDC);
    CRect rect;
    HICON hIcon;
    COLORREF crText = m_crText;
    // 绘制彩色矩形部分
    rect.CopyRect(&lpDIS->rcItem);

    // 绘制弹起/按下/聚焦/禁止状态

```

```

UINT action = lpDIS->itemAction;
UINT state = lpDIS->itemState;
CString strText;
LOGFONT lf;
lf = m_lf;

CFont dispFont;
CFont *pFont;
if (lpDIS->itemData != NULL)
{
    strText = (((MENUDATA *) (lpDIS->itemData))>menuText);
    if (((((MENUDATA *) (lpDIS->itemData))>menuIconNormal) == -1)
        hIcon = NULL;
    else if (state & ODS_SELECTED)
    {
        if (((((MENUDATA *) (lpDIS->itemData))>menuIconSelected) !=
            -1)
            hIcon = AfxGetApp()>LoadIcon (((MENUDATA *) (lpDIS->
                itemData))>menuIconSelected);
        else
            hIcon = AfxGetApp()>LoadIcon (((MENUDATA *) (lpDIS->
                itemData))>menuIconNormal);
    }
    else
        hIcon = AfxGetApp()>LoadIcon (((MENUDATA *) (lpDIS->itemDa-
            ta))>menuIconNormal);
}
else
{
    strText.Empty();
    hIcon = NULL;
}

if ((state & ODS_SELECTED))
{
    // 绘制下边界
    CPen *pOldPen = pDC->SelectObject (&m_penBack);
    // 只需要将文本高亮显示
    if (m_hilightStyle != Normal)
    {
        pDC->FillRect (rect,&m_brBackground);
    }
}

```

```

else
{
    pDC->FillRect (rect,&m_brSelect);
}

pDC->SelectObject (pOldPen);
pDC->Draw3dRect (rect,GetSysColor
    (COLOR_3DHILIGHT),GetSysColor(COLOR_3DSHADOW));
lf.lfWeight = FW_BOLD;
if ((HFONT)dispFont != NULL)
    dispFont.DeleteObject ();
dispFont.CreateFontIndirect (&lf);
crText = m_clrHilightText;
}
else
{
    CPen *pOldPen = pDC->SelectObject (&m_penBack);
    pDC->FillRect (rect,&m_brBackground);
    pDC->SelectObject (pOldPen);
    // 绘制上边界
    pDC->Draw3dRect (rect,m_clrBack,m_clrBack);
    if ((HFONT)dispFont != NULL)
        dispFont.DeleteObject ();
    dispFont.CreateFontIndirect (&lf); //Normal
}

// 绘制文本
if (hIcon != NULL)
    DrawIconEx (pDC->GetSafeHdc(),rect.left,rect.top,hIcon,
        (m_iconX)? m_iconX:32,(m_iconY)? m_iconY:32,0,NULL,DI_NOR-
        MAL)

// 平移以留出复选标记位置
rect.left = rect.left + ((m_iconX)? m_iconX:32);

if (! strText.IsEmpty())
{
    int iOldMode = pDC->GetBkMode();
    pDC->SetBkMode( TRANSPARENT);
    pDC->SetTextColor( crText);
    pFont = pDC->SelectObject (&dispFont);
    pDC->DrawText (strText,rect,DT_LEFT|DT_SINGLELINE|DT_VCENTER);
    pDC->SetBkMode( iOldMode );
    pDC->SelectObject (pFont); //设置为原来的字体
}

```

```
dispFont.DeleteObject ();  
}
```

函数首先得到菜单项的状态,然后根据其状态选择合适的图标,并将其绘制在菜单项左侧。无论实际的绘制操作多么复杂,只要读者把握自绘制的基本原则,就一定能够解决。

TCMenu 类中的其他成员函数主要是执行一些辅助功能,请读者自行参照配套光盘中 chap8/iconmenu 目录下的源代码。

本章小结

本章主要介绍的是菜单编程的基本思路和方法。通过本章的学习,读者应该达到以下几点:

- 掌握 CMenu 类的使用。
- 掌握不同类型菜单的编程方法。

第 8 章 工 具 栏

工具栏为用户提供了执行菜单命令的更为快捷、直观的方式。本章将向读者介绍如何定制自己的工具栏。

本章要点：

- CToolBar 类的使用；
- 标准工具栏的使用；
- 创建 IE 风格的工具栏；
- 创建下拉式工具栏；
- 在工具栏中使用控件；
- 定制和排列工具栏；
- 在对话框中使用工具栏和工具提示；
- 在 MDI 应用程序中切换工具栏。

8.1 工具栏编程基础

菜单是由 CToolBar 类进行管理的,图 8-1 所示为 CToolBar 类的派生结构。

形象地说 CToolBar 对象类似于包括一行位图按钮(有时还有间隔)的控件栏。其中按钮的行为可以为下压式按钮、复选框或单选按钮。一般来说,CToolBar 对象是被嵌入框架窗口类中(CFrameWnd 或 CMDIFrameWnd 类或其派生类)使用的。在 MFC 4.0 中新引入的 CToolBar::GetToolBarCtrl 函数,提供了获得 Windows 为工具栏添加的扩展功能的手段。

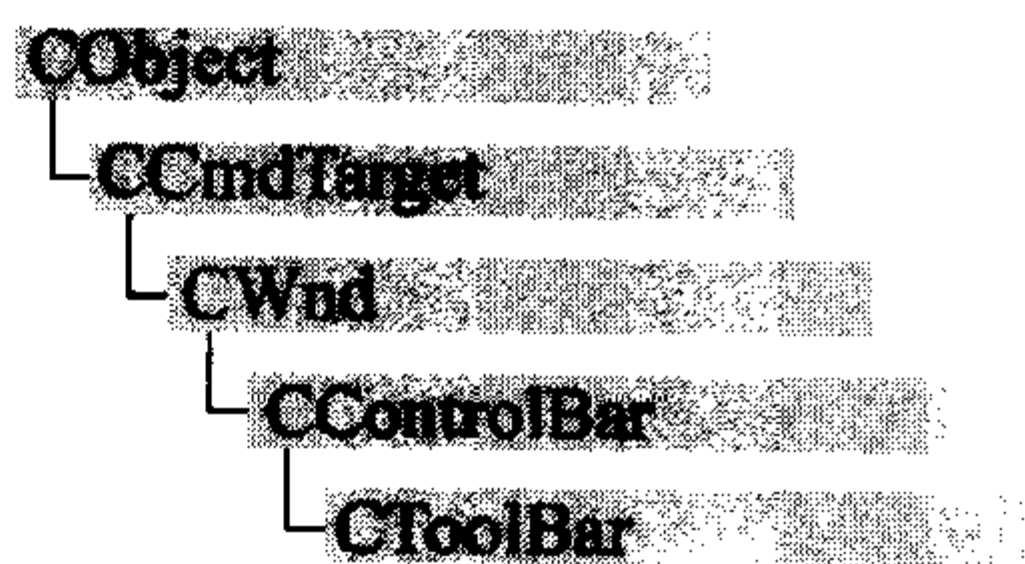


图 8-1 CToolBar 类的派生结构

8.1.1 工具栏概述

Visual C++ 提供了两种创建工具栏的方法。在资源编辑器中创建工具栏的步骤如下：

- (1) 创建工具栏资源。
- (2) 构造 CToolBar 对象。
- (3) 调用 Create 或 CreateEx 函数以创建 Windows 工具栏,并将其与 CToolBar 相连接。
- (4) 调用 LoadToolBar 函数以载入工具栏资源。

此外,还可以使用如下步骤在代码中直接创建：

- (1) 构造 CToolBar 对象。
- (2) 调用 Create 或 CreateEx 函数以创建 Windows 工具栏,并将其与 CToolBar 相连接。
- (3) 调用 LoadBitmap 函数以载入工具栏按钮位图。
- (4) 调用 SetButtons 设置工具栏按钮的风格,并将按钮和位图一一对应。

所有工具栏按钮的图像都是从载入的位图中得到的,当然这个位图必须包括每个按钮所需的图像。这些按钮图像的尺寸必须一致,默认的按钮图像尺寸为宽 16 像素,高 15 像素;此外按钮图像在位图中必须连续排列。这就要求在选择工具栏位图时要非常细心。

SetButtons 函数将包括按钮 ID 的数组指针,和将设置的按钮数目作为参数。它将设置每个工具栏按钮的 ID(数组中的相应元素),并为每个按钮分配图像索引,也就是指定按钮图像在工具栏位图中的位置。如果数组中的某个值为 ID_SEPARATOR,则表示该元素对应工具栏中的分隔符,函数不为其分配图像索引。

一般来说,位图中的图像顺序就是在屏幕上绘制的顺序。不过,可以通过调用 SetButtonInfo 函数改变图像顺序和绘制顺序之间的关系。

工具栏中的所有按钮都具有相同的尺寸,默认尺寸为 24 × 22 像素。根据 Windows 用户界面设计的一般规定,按钮图像和按钮尺寸之间的差额被作为按钮的边框。

每个按钮都有相应的图像,并且其不同状态(按下、弹起、禁止、按下禁止等)下的外观都由同一个按钮图像生成。虽然位图可以包含任何颜色,然而黑色和灰色是最好的选择。

默认情况下,工具栏按钮为下压按钮风格。不过,它也可以被设置为复选按钮或单选按钮。复选按钮有三个状态:检查、清除和不确定态。单选按钮只有两种状态:检查和清除态。

如果需要检索或单独设置某个按钮或间隔的风格,则可以使用 GetButtonStyle 函数得到相应按钮的风格,或调用 SetButtonStyle 函数设置其新风格。需要动态改变按钮风格时,SetButtonStyle 函数尤其有用。

如果得到或设置按钮上显示的文本,则可以调用 GetButtonText 和 SetButtonText 函数来完成。如果要创建一个复选按钮,则应该将其风格指定为 TBBS_CHECKBOX,或在 ON_UPDATE_COMMAND_UI 消息的相应函数中,调用 CcmdUI 对象的 SetCheck 成员函数。

要创建单选按钮,则可以在 ON_UPDATE_COMMAND_UI 消息的相应函数中,调用 CcmdUI 对象的 SetRadio 成员函数。如果需要实现成组单选按钮,则对所有组内按钮都必须在 ON_UPDATE_COMMAND_UI 消息处理函数中调用 SetRadio 成员函数。

使用 CToolBar 类能够在应用程序中载入、创建工具栏,改变工具栏的外观和特性,下面给出该类中常用的成员函数。

8.1.2 构造函数

CToolBar 类的构造函数包括:CToolBar、Create、CreateEx、SetSizes、SetHeight、LoadToolBar、LoadBitmap、SetBitmap 和 SetButtons,它们可以完成构造工具栏对象、设置工具栏属性等操作。

- CToolBar

调用该函数以构造 CToolBar 对象,其原型为:

```
CToolBar( );
```

- Create

调用该函数将创建工具栏并将其连接到一个工具栏对象,其原型如下:

```
BOOL Create( CWnd* pParentWnd, DWORD dwStyle = WS_CHILD|WS_VISIBLE|CBRS_TOP,
UINT nID = AFX_IDW_TOOLBAR );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

pParentWnd ——指定了工具栏所属的框架窗口。

dwStyle ——指定了工具栏的风格,其取值可为表 8-1 中一项或几项的联合。

表 8-1 常用的工具栏风格

风格常量	描述
CBRS_TOP	工具栏位于框架窗口的顶部
CBRS_BOTTOM	工具栏位于框架窗口的底部
CBRS_NOALIGN	父窗口改变尺寸时,工具栏不改变位置
CBRS_TOOLTIPS	鼠标滑过时,工具栏显示工具提示
CBRS_SIZE_DYNAMIC	工具栏可变
CBRS_SIZE_FIXED	工具栏固定
CBRS_FLOATING	工具栏可浮动
CBRS_FLYBY	状态栏显示工具栏按钮信息
CBRS_HIDE_INPLACE	工具栏不显示

nID ——指定了工具栏 ID。

- CreateEx

调用该函数将按扩展风格创建工具栏并将其连接到一个工具栏对象,其原型如下:

```
BOOL CreateEx(CWnd* pParentWnd, DWORD dwCtrlStyle = TBSTYLE_FLAT, DWORD dw-
Style = WS_CHILD | WS_VISIBLE | CBRS_ALIGN_TOP, CRect rcBorders = CRect(0, 0,
0, 0), UINT nID = AFX_IDW_TOOLBAR);
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

pParentWnd ——指定了工具栏所属的框架窗口。

dwCtrlStyle ——指定了工具栏的扩展风格,该参数的默认值为 TBSTYLE_FLAT。

dwStyle ——指定了工具栏的风格,其取值参见表 5-8 中一项或几项的联合。

rcBorders ——指定了工具栏窗口的边框宽度。

nID ——指定了工具栏 ID。

使用该函数可以创建具有 IE 风格的工具栏按钮。

- LoadBitmap

调用该函数以载入将作为工具栏的位图资源,其原型如下:

```
BOOL LoadBitmap( LPCTSTR lpszResourceName );  
BOOL LoadBitmap( UINT nIDResource );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

lpszResourceName ——指定了位图资源名称。

nIDResource ——指定了位图资源 ID。

位图中应该包含与工具栏按钮数目相同的图形。如果图形不是标准尺寸(16 × 15),那么可以调用 SetSizes 函数来设置合适的按钮和图形尺寸。

- LoadToolBar

调用该函数以载入工具栏资源,其原型如下:

```
BOOL LoadToolBar( LPCTSTR lpszResourceName );  
BOOL LoadToolBar( UINT nIDResource );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

lpszResourceName ——指定了位图资源名称。

nIDResource ——指定了位图资源 ID。

- SetBitmap

调用该函数为工具栏指定位图资源,其原型如下:

```
BOOL SetBitmap( HBITMAP hbmImageWell );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

hbmImageWell ——指定了与工具栏相联系的位图句柄。可以调用该函数来改变按钮图像。

- SetButtons

调用该函数设置工具栏按钮的 ID,其原型如下:

```
BOOL SetButtons( const UINT* lpIDArray, int nIDCount );
```

参数:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

lpIDArray ——指定了将为工具栏按钮设置的 ID。

nIDCount ——指定了将设置 ID 的按钮数目。该函数将工具栏中的各个按钮 ID 设置为 lpIDArray 结构中的相应元素。该函数也将每个按钮的风格设置为 TBBS_BUTTON, 将每个间隔的风格被设置为 TBBS_SEPARATOR, 并且将设置图形索引。图形索引指定了按钮图形在位图中的位置。

用户不需要考虑间隔, 因为该函数将不为间隔分配图形索引。例如如果用户的工具栏按钮处于 0、1 和 3 位, 而 2 位为间隔, 那么在位图中 0、1 和 2 位的图形将被分别分配给按钮 0、1 和 3。如果 lpIDArray 为 NULL, 则函数将为 nIDCount 所指定的数目分配位置, 使用 SetButtonInfo 可以设置每个按钮的属性。

- SetHeight

调用该函数设置工具栏的高度, 其原型如下:

```
void SetHeight( int cyHeight );
```

参数:

cyHeight ——指定了将为工具栏设置的高度。

在调用 SetSizes 函数后, 使用 SetHeight 函数以修改标准的工具栏高度。如果高度太小, 则按钮将被裁剪。如果不调用该函数, 则框架将根据按钮的高度决定工具栏的高度。

- SetSizes

调用该函数设置工具栏按钮大小, 其原型如下:

```
void SetSizes( SIZE sizeButton, SIZE sizeImage );
```

参数:

sizeButton ——指定了工具栏按钮的尺寸。

sizeImage ——指定了工具栏位图中按钮图像的尺寸。

工具栏按钮的尺寸必须至少比位图按钮尺寸在宽度上大 7 像素, 在高度上大 6 像素。该函数同时也设置了工具栏按钮的高度。

8.1.3 属性操作函数

CToolBar 类的属性操作函数包括: CommandToIndex、GetButtonInfo、SetButtonStyle、GetButtonStyle、SetButtonInfo、GetItemId、GetButtonText、SetButonText 和 GetToolBarCtrl, 它们可以完成检索和设置工具栏风格、ID 等操作。

- CommandToIndex

调用该函数以获得给定 ID 值的命令按钮的工具栏索引, 其原型如下:

```
int CommandToIndex( UINT nIDFind );
```

返回值:

如果函数调用成功, 则返回按钮的索引。如果返回 -1, 则表示没有为该按钮指定 ID。

参数:

nIDFind ——指定了为工具栏按钮的命令 ID。

- GetButtonInfo

调用该函数以获得给定工具栏按钮的 ID、风格与索引,其原型如下:

```
void GetButtonInfo( int nIndex, UINT& nID, UINT& nStyle, int& iImage ) const;
```

参数:

nIndex ——指定了将获取其信息的工具栏按钮或间隔的索引。

nID ——将返回按钮的命令 ID。

nStyle ——将返回按钮的风格。

iImage ——将返回按钮位图索引。如果 nIndex 参数指定的是一个间隔,则 iImage 将返回间隔的像素宽度。

- GetButtonStyle

调用该函数以获得工具栏按钮的风格,其原型如下:

```
UINT GetButtonStyle( int nIndex ) const;
```

返回值:

如果函数调用成功,则返回由 nIndex 指定的按钮或间隔的风格。

参数:

nIndex ——指定了要获取其风格的工具栏按钮或间隔的索引。

- GetButtonText

调用该函数以获得工具栏按钮的文本,其原型如下:

```
CString GetButtonText( int nIndex ) const;  
void GetButtonText( int nIndex, CString& rString ) const;
```

返回值:

如果函数调用成功,则返回包含指定按钮文本的 CString 对象。

参数:

nIndex ——指定了要获取其文本的工具栏按钮或间隔的索引。

rString ——将返回工具栏按钮的文本。

- GetItemID

调用该函数以获得给定索引的工具栏按钮的 ID,其原型如下:

```
UINT GetItemID( int nIndex ) const;
```

返回值:

如果函数调用成功,则返回指定按钮的 ID。

参数:

nIndex ——指定了要获取其命令 ID 的工具栏按钮或间隔的索引。

- GetToolBarCtrl

调用该函数以获得工具栏控制对象指针,其原型如下:

```
CToolBarCtrl& GetToolBarCtrl( ) const;
```

返回值:

如何函数调用成功,则返回对应的 CToolBarCtrl 对象指针。

使用 GetToolBarCtrl 函数可以获得工具栏的控制对象,因而可以使用该控制对象所提供的高级函数来优化应用程序的工具栏显示和功能。

- GetItemRect

调用该函数以获得工具栏按钮的客户区矩形,其原型如下:

```
virtual void GetItemRect( int nIndex, LPRECT lpRect );
```

参数:

nIndex ——指定了要获取其矩形尺寸的工具栏按钮或间隔的索引。

lpRect ——将返回按钮矩形尺寸。

- SetButtonInfo

调用该函数以设置给定工具栏按钮的 ID、风格与索引,其原型如下:

```
void SetButtonInfo( int nIndex, UINT nID, UINT nStyle, int iImage );
```

参数:

nIndex ——指定了将设置其信息的工具栏按钮索引。

nID ——指定了将为按钮设置的命令 ID。

nStyle ——指定了将为按钮设置的风格,其取值参见表 8-1。

iImage ——指定了将为按钮设置的位图索引。

- SetButtonStyle

调用该函数设置工具栏按钮的风格,其原型如下:

```
void SetButtonStyle( int nIndex, UINT nStyle );
```

参数:

nIndex ——指定了将设置其风格的按钮索引。

nStyle ——指定了将为按钮设置的风格,其取值参见表 8-1。

- SetButtonText

调用该函数设置工具栏按钮的文本,其原型如下:

```
BOOL SetButtonText( int nIndex, LPCTSTR lpszText );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

nIndex ——指定了将设置其文本的按钮索引。

lpszText ——指定了将为按钮设置的文本。

8.2 使用标准工具栏

在应用程序中使用标准工具栏一般要经过以下步骤：

- (1) 编辑工具栏资源。
- (2) 为每个工具栏按钮增加消息处理函数。
- (3) 将工具栏载入应用程序中。

编辑工具栏资源和为工具栏按钮增加消息处理函数都很简单,通过资源编辑器和 ClassWizard 能够非常容易的完成,需要注意的是,由于有些工具栏按钮命令是菜单命令的另一个入口,因此其 ID 和消息处理函数都与相应的菜单命令相同。

下面主要向读者介绍如何将工具栏载入应用程序中。应用程序一般通过重载 CMainFrame 类的 OnCreate 函数载入工具栏。如果应用程序中包括不同的子框架窗口类型,需要为每个子框架窗口创建不同的工具栏,这时也可在 CChildFrame 类的 OnCreate 函数中载入工具栏。OnCreate 函数是 WM_CREATE 消息映射的处理函数,该消息处理函数将在框架窗口创建时被调用。载入工具栏的一般步骤如下:

- (1) 声明工具栏对象。

```
class CMainFrame : public CMDIFrameWnd
{
    ...
    CToolBar m_wndToolBar;    //标准工具栏
    CToolBar m_DrawToolBar;   //绘图工具栏
    ...
};
```

- (2) 创建工具栏对象,载入相应的工具栏资源,如清单 8-1 所示:

清单 8-1 OnCreate()函数

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    ...
    //载入标准工具栏
    if (! m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE |
        CBRS_TOP | CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_
        DYNAMIC) || ! m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
    {
        TRACE0("创建标准工具栏失败\n");
        return -1;    // fail to create
    }

    //使标准工具栏浮动
    m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
    EnableDocking(CBRS_ALIGN_ANY);
    DockControlBar(&m_wndToolBar);
}
```

```
//载入绘图工具栏
if (! m_DrawToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE |
    CBRS_ALIGN_LEFT | CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_
    SIZE_DYNAMIC) || ! m_DrawToolBar.LoadToolBar(IDR_DRAW_TOOLBAR))
{
    TRACE0("创建绘图工具栏失败\n");
    return -1;
}

//使绘图工具栏可停靠
m_DrawToolBar.EnableDocking(CBRS_ALIGN_ANY);
EnableDocking(CBRS_ALIGN_ANY);
DockControlBar(&m_wndToolBar);
...
}
```

这样就将工具栏载入了应用程序中,当程序运行时选择工具栏按钮就会启动相应的消息处理函数,从而使应用程序作出响应。程序中使用了 `EnableDocking` 和 `DockControlBar` 两个函数以使工具栏停靠,实际上这就是在 Windows 应用程序中经常看到的能够用鼠标拖动的工具栏。

如果没有为工具栏按钮添加消息处理函数,那么在窗口创建后,该工具栏按钮将呈灰色,不能使用。上面介绍的是最常用的创建工具栏的方法,除此之外,还可以通过位图或直接声明工具栏按钮 ID 数组来创建,这些方法将在以后的程序中陆续向读者介绍。

处理菜单更新的更新命令 UI 消息处理函数同样也可以用于工具栏按钮,如果更新命令的 UI 消息处理函数用 `FALSE` 参数来调用 `CCmdUI` 类的 `Enable` 成员函数,那么相应的按钮就会被禁止,从而不会再对按钮的鼠标单击作出响应。

对于菜单项来说, `CCmdUI` 的 `SetCheck` 函数将设置菜单项的检查标记,而对于工具栏按钮来说,该函数则把它所包含的按钮都当成检查框按钮来处理了。如果更新命令 UI 消息处理函数以参数 1 来调用 `SetCheck` 函数,那么相应的按钮就会被设置为按下(检查)状态,而如果更新命令 UI 消息处理函数以参数 0 来调用该函数,那么相应的按钮就会被设置为弹起状态(未被检查)。

8.3 创建 IE 风格的工具栏

经常上网的读者应该对 IE 浏览器非常熟悉了,那么您是否注意过其独具特色的工具栏呢? 本节就向读者介绍其实现方法。

8.3.1 使工具栏具有“热敏”变色风格

当鼠标从 IE 工具栏按钮上掠过时,它在凸起(这一点与普通平面工具栏一致)的同时,按钮图案的颜色也相应改变。我们将其称为“热敏”变色,其实现步骤如下:

(1) 在资源编辑器中插入两个位图,它们除了图案颜色不同外应该完全一致。将其 ID 分别设置为: IDB_TOOLBAR_COLD 和 IDB_TOOLBAR_HOT。其中 IDB_TOOLBAR_COLD (“冷”工具栏)的图案颜色应该较暗,例如为灰色,用于表示工具栏的正常状态;而 IDB_TOOLBAR_HOT 的图案应该较亮,例如为红色,用于表示鼠标掠过时的状态(“热”工具栏)。

(2) 在 CMainFrame::OnCreate 函数(或在其他创建工具栏的函数中)中定制工具栏风格,示范代码如下所示:

```
// 设置“热”工具栏图像列表
CImageList imageList;
CBitmap bitmap;

// 创建并设置正常工具栏图像列表
bitmap.LoadBitmap(IDB_TOOLBAR_COLD);
imageList.Create(21, 20, ILC_COLOR32|ILC_MASK, 13, 1);
imageList.Add(&bitmap, RGB(255,0,255));
m_wndBrowserBar.SendMessage(TB_SETIMAGELIST, 0, (LPARAM)imageList.m_hImageList);
imageList.Detach();
bitmap.Detach();

// 创建并设置“热”工具栏图像列表
bitmap.LoadBitmap(IDB_TOOLBAR_HOT);
imageList.Create(21, 20, ILC_COLOR32|ILC_MASK, 13, 1);
imageList.Add(&bitmap, RGB(255,0,255));
m_wndBrowserBar.SendMessage(TB_SETHOTIMAGELIST, 0,
                             (LPARAM)imageList.m_hImageList);
imageList.Detach();
bitmap.Detach();
```

其中 m_wndBrowserBar 为工具栏对象。上述代码在设置“冷”、“热”工具栏时使用的方法是发送 TB_SETIMAGELIST 和 TB_SETHOTIMAGELIST 消息。实际上这个操作已经被封装在 CToolBarCtrl 类的 SetImageList 和 SetHotImageList 函数中了,因此上述代码也可以修改为如下形式:

```
...
m_wndBrowserBar.GetToolBarCtrl().SetHotImageList(&imageList);
...
m_wndBrowserBar.GetToolBarCtrl().SetImageList(&imageList);
```

8.3.2 在工具栏中显示文本

IE 工具栏的另一个特点就是每个按钮都带有说明文字。在上一节中已经创建了工具栏位图,而要使工具栏能够生效还必须为每个按钮设置 ID。这一般是在 CMainFrame 类的源文件开始处进行设置,示范代码如下:

```
static UINT BrowserBar[] =
```

```

    {
        ID_GO_BACK,
        ID_GO_FORWARD,
        ID_VIEW_STOP,
        ID_VIEW_REFRESH,
        ID_GO_START_PAGE,
        ID_GO_SEARCH,
        ID_FONT,
    };
};

```

这实际为 `m_wndBrowserBar` 对象指定了 6 个按钮 ID,也就是说该工具栏由 6 个按钮组成。因此在为每个按钮指定 ID 前,应该调用 `SetButtons` 函数设置工具栏包含按钮数目,示范代码如下:

```
m_wndBrowserBar.SetButtons(BrowserBar, sizeof(WriteBar)/sizeof(UINT));
```

接下来就可以为每个按钮指定文本了,调用 `SetButtonText` 能够完成这一操作,示范代码如下:

```

// 设置工具栏文本
m_wndBrowserBar.SetButtonText(0, _T("后退"));
m_wndBrowserBar.SetButtonText(1, _T("前进"));
m_wndBrowserBar.SetButtonText(2, _T("停止"));
m_wndBrowserBar.SetButtonText(3, _T("刷新"));
m_wndBrowserBar.SetButtonText(4, _T("主页"));
m_wndBrowserBar.SetButtonText(5, _T("搜索"));
m_wndBrowserBar.SetButtonInfo(6, ID_FONT, TBSTYLE_BUTTON | TBSTYLE_DROPDOWN,
6);
m_wndBrowserBar.SetButtonText(6, _T("字体"));

```

经过上述操作,就能够创建出 IE 风格的工具栏了。

8.4 创建下拉菜单式工具栏按钮

使用 `SetButtonInfo` 函数将工具栏按钮的风格设置为 `TBSTYLE_DROPDOWN` 即可创建下拉式工具按钮,下面的示范代码将工具栏中第 5 个按钮设置为下拉式风格,注意工具栏间隔也算一个按钮。

```

m_wndToolBar.GetToolBarCtrl().SetExtendedStyle(TBSTYLE_EX_DRAWDDARROWS);
m_wndToolBar.SetButtonStyle(5, TBSTYLE_BUTTON | TBSTYLE_DROPDOWN);

```

在绘图应用程序中,当用户按下缩放观察比例命令按钮时,就会打开一个菜单,该菜单中包括视图能够被缩放的比例。此时按钮向主框架窗口传递的并非一般按钮的 `WM_COMMAND` 消息,而是通告下拉操作的 `TBN_DROPDOWN` 消息。第 2 章已向读者介绍过,框架窗口所能处理的工具栏按钮消息只是 `WM_COMMAND` 和 `UPDATE_COMMAND_UI`

消息,因此要想使框架窗口对按钮的下拉操作作出处理的话,必须使用别的途径。

MFC 中提供了 ON_NOTIFY 宏用于处理复合对象消息,实际上,用户对对象的每个操作都会导致系统向框架窗口发送 WM_NOTIFY 消息。WM_NOTIFY 消息的 wParam 参数包含了发送消息的控件 ID, lParam 参数则存放了一个结构指针。该结构指针可以指向一个 NMHDR 结构,但也可以指向一个较大的结构,但这个较大结构的第一个成员也是 NMHDR 结构。由于 lParam 参数的不确定性,用户可以使用不同的结构类型来将 lParam 转化为需要使用的结构指针。

在绝大多数情况下, lParam 指向的是一个较大的结构。因此在使用时通常需要转化数据类型。只有对于很少的通告消息例如 TTN_SHOW 或 TTN_TOP 等,才确实使用 NMHDR 结构指针。

在 NMHDR 结构中包含了控件的 ID 和通告消息(例如 TTN_SHOW), NMHDR 结构的定义如下:

```
typedef struct tagNMHDR {
    HWND hwndFrom;    //通告消息来自的窗口句柄
    UINT idFrom;      //通告消息来自的控件 ID
    UINT code;        //通告消息
} NMHDR;
```

对于 TTN_SHOW 消息,结构的 code 成员就是 TTN_SHOW。不过大多数通告消息都会传递一个较大的结构,该结构将 NMHDR 作为结构的第一个成员。例如,对于在列表视图控件中按下某个键时发送的 LVN_KEYDOWN 通告消息, lParam 中包含的将是 LV_KEYDOWN 结构,该结构的定义如下:

```
typedef struct tagLV_KEYDOWN {
    NMHDR hdr;
    WORD wVKey;
    UINT flags;
} LV_KEYDOWN;
```

由于 NMHDR 是结构的第一个成员,因此该结构既可以作为 LV_KEYDOWN 使用也可以作为 NMHDR 使用。

ON_NOTIFY 消息映射用于处理 WM_NOTIFY 消息。该消息映射在默认情况下,将检查处理该消息映射的函数。一般来说,用户不需要重载 OnNotify 函数。相反,用户需要提供一个处理函数并且需要将该处理函数添加到应用程序的消息映射中。

使用 ClassWizard 可以为应用程序添加 ON_NOTIFY 消息映射入口,以及骨架处理函数。ON_NOTIFY 消息映射宏的原型如下:

```
ON_NOTIFY( wNotifyCode, id, memberFxn )
```

其中 wNotifyCode 为将处理的通告消息,例如 LVN_KEYDOWN。参数 id 为发送消息的控件 ID。参数 memberFxn 为处理通告消息的函数。该函数必须使用如下的定义形式:

```
afx_msg void memberFxn( NMHDR * pNotifyStruct, LRESULT * result );
```

其中参数 pNotifyStruct 为指向通告消息的结构,该结构定义如前所示。参数 result 为

指向在返回前将设置的结果码指针。

清单 8-2 列出了 Painter 中处理缩放下拉式工具按钮的代码。

清单 8-2 OnDropDown()函数

```
void PainterView::OnDropDown(NMHDR * pNotifyStruct, LRESULT * pResult)
{
    NMTOOLBAR * pNMToolBar = (NMTOOLBAR *)pNotifyStruct;
    CRect rect;
    CMainFrame * pWnd = (CMainFrame *)AfxGetMainWnd();

    // 将工具栏按钮的矩形转换为屏幕坐标,以便确定从何处打开下拉菜单
    pWnd->m_wndToolBar.GetToolBarCtrl().GetRect(pNMToolBar->iItem, &rect);
    rect.top = rect.bottom;
    ::ClientToScreen(pNMToolBar->hdr.hwndFrom, &rect.TopLeft());
    if(pNMToolBar->iItem == ID_ENLARGE)
    {
        CMenu menuR;
        CMenu * pPopupR;

        // 从资源中载入菜单
        menuR.LoadMenu(IDR_ZOOM_POPUP);
        pPopupR = menuR.GetSubMenu(0);
        pPopupR->TrackPopupMenu(TPM_LEFTALIGN | TPM_LEFTBUTTON, rect.left,
            rect.top + 1, AfxGetMainWnd());
    }
    *pResult = TBDDRET_DEFAULT;
}
```

此外还需在类的头文件中手动添加如下代码:

```
afx_msg void OnDropDown(NMHDR * pNotifyStruct, LRESULT * pResult);
```

该代码必须处于 DECLARE_MESSAGE_MAP 语句之前,然后在类的源文件中添加如下代码:

```
ON_NOTIFY(TBN_DROPDOWN, AFX_IDW_TOOLBAR, OnDropDown)
```

该代码必须处于 BEGIN_MESSAGE_MAP 和 END_MESSAGE_MAP 语句之间。图 8-2 所示即为按下“缩放”按钮时出现的下拉式菜单。

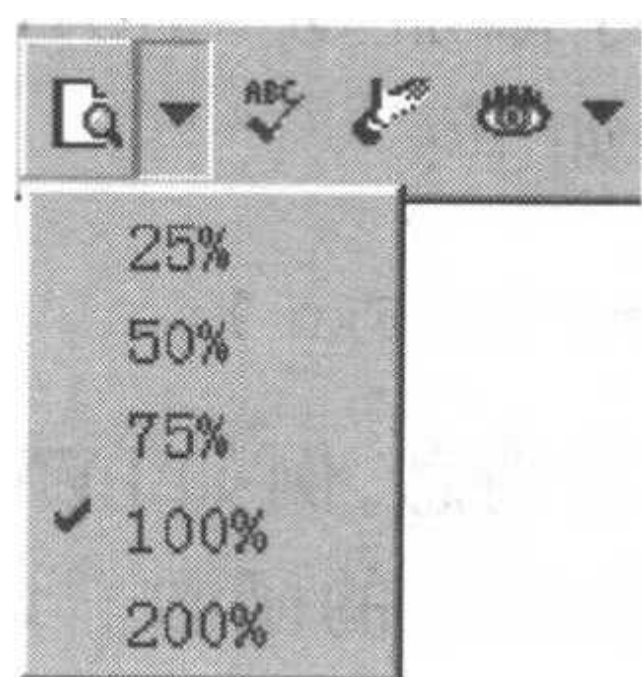


图 8-2 下拉菜单式工具栏按钮

8.5 在工具栏中使用控件

在工具栏中加入控件不仅能够为其添加附加功能,而且能够改善界面的外观。然而使用工具栏资源编辑器只能向工具栏中添加按钮和分隔,因此加入其他控件需要定制代码实现。本节中将给出两个例子,分别向控件中加入组合框和一组复选框。

8.5.1 添加组合框控件

(1) 首先向工具栏资源中添加一个按钮,这个按钮将被用来创建组合框。用分隔也可以实现,但是那样会使组合框与旁边的按钮紧挨在一起,不太美观;而使用按钮,则可以在创建时就在该按钮两边设置间隔,这样创建的组合框就不会与两边的工具按钮挨在一起了。如果一定要使用间隔来创建,而又不使所创建的控件与相邻工具栏按钮挨在一起,就必需像创建状态栏那样首先定义一个工具栏 ID 数组,例如连续定义 3 个间隔,然后使用中间的间隔创建控件即可。这里将使用前一种方法,并将占位按钮的 ID 设置为 IDP_PLACEHOLDER2,如图 8-3 所示。



图 8-3 包含占位按钮的工具栏资源

(2) 创建 CToolBar 的派生类 CMainToolBar,用来管理包含其他控件的工具栏对象。并在类中声明代表该控件的变量。对于组合框来说,变量应该为 CComboBox 类型。在该类中并不需要添加其他函数。

```
CComboBox m_wndSnap;
```

(3) 在 CMainFrame 类中将工具栏对象声明为 CMainToolBar 类型:

```
CMainToolBar m_wndToolBar;GC
```

(4) 在 CMainFrame 类中的 OnCreate 函数中将占位按钮替换为组合框,实现代码如清单 8-3 所示:

清单 8-3 OnCreate()函数

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
```

```

#define SNAP_WIDTH 80 //组合框的宽度
//首先得到工具栏占位按钮的索引
index = 0;
while(m_wndToolBar.GetItemID(index) != IDP_PLACEHOLDER2) index++;

//将按钮转换为分隔符,并取得其位置
m_wndToolBar.SetButtonInfo(index, IDP_PLACEHOLDER2, TBBS_SEPARATOR, SNAP_WIDTH);
m_wndToolBar.GetItemRect(index, &rect);

//延展矩形以允许组合框有空间显示下拉列表
rect.top += 2;
rect.bottom += 200;

// 然后创建并显示组合框
if (! m_wndToolBar.m_wndSnap.Create(WS_CHILD|WS_VISIBLE | CBS_AUTOHSCROLL
    | CBS_DROPDOWNLIST | CBS_HASSTRINGS, rect, &m_wndToolBar, IDC_SNAP_COMBO))
{
    TRACE0("Failed to create combo-box\n");
    return FALSE;
}
m_wndToolBar.m_wndSnap.ShowWindow(SW_SHOW);

//将选项填充到组合框中
m_wndToolBar.m_wndSnap.AddString("SNAP OFF");
m_wndToolBar.m_wndSnap.AddString("SNAP GRID");
m_wndToolBar.m_wndSnap.AddString("SNAP RASTER");
m_wndToolBar.m_wndSnap.AddString("SNAP VERTEX");
m_wndToolBar.m_wndSnap.AddString("SNAP LINE");
m_wndToolBar.m_wndSnap.SetCurSel(0);
}

```

而使用工具栏中的组合框,与在对话框中使用组合框的方法相同,即为该组合框添加相应的标准 Windows 消息映射即可。创建后的工具栏效果如图 8-4 所示。

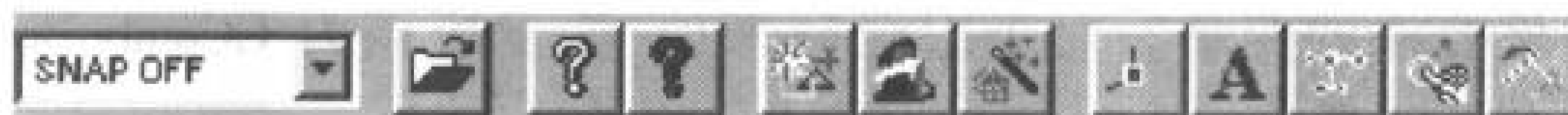


图 8-4 加入组合框控件后的工具栏效果

8.5.2 添加复选框控件

在本节中将 4 个复选框添加到工具栏中。这些控件也将替代工具栏资源中的占位按钮。不过需要注意的是,要使这些控件使用较小的字体。

(1) 首先向工具栏资源中添加一个按钮,这个按钮将被用来创建复选框,并在该按钮两边设置间隔,然后将占位按钮的 ID 设置为 IDP_PLACEHOLDER2。

(2) 创建 CToolBar 的派生类 CCoupleToolBar,用来管理包含复选框控件的工具栏对象。并在类中声明代表该控件的变量。在该类中并不需要添加其他函数。


```

CButton m_wndCenter;
CButton m_wndEdge;
CButton m_wndTrack;
CButton m_wndZoom;

```

(3) 在 CMainFrame 类中将工具栏对象声明为 CCoupleToolBar 类型,同时还添加小字体变量:

```

CMainToolBar m_wndToolBar;
Cfont gSmallFont;

```

(4) 在 CMainFrame 类的 OnCreate 函数中将占位按钮替换为组合框,实现代码如清单 8-4 所示:

清单 8-4 OnCreate()函数

```

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    // 设置小字体
    CClientDC DC(GetDesktopWindow());
    long logical_pixels = DC.GetDeviceCaps(LOGPIXELSX);
    if(logical_pixels < 100)
    {
        gSmallFont.CreatePointFont(67,"DEFAULT",NULL);
    }
    else
    {
        gSmallFont.CreatePointFont(50,"DEFAULT",NULL);
    }

    // 创建 4 个组合框
    #define CHECK_WIDTH 94
    int index;
    CRect rect;
    CRect safe_rect;
    index = 0;

    while(m_wndViewBar.GetItemID(index) != IDP_PLACHOLDER)
        index++;

    // 创建复选框
    m_wndViewBar.SetButtonInfo(index, IDP_PLACHOLDER, TBBS_SEPARATOR, CHECK_WIDTH);
    m_wndViewBar.GetItemRect(index, &rect);
    safe_rect = rect;
    rect.left += 2;
    rect.right = rect.left + ((CHECK_WIDTH / 2) - 4);
    rect.top = 2;
    rect.bottom = rect.top + 10;
    if (! m_wndViewBar.m_wndCenter.Create("CNTR", BS_CHECKBOX | WS_CHILD | WS_VISIBLE, rect, &m_wndViewBar, IDM_COUPLE))

```


8.6 使用 16M 色位图创建工具栏

Visual C++ 的资源管理器只能处理 256 色的位图。也就是说在默认情况下,工具栏无法显示高于 256 色的按钮图案。这在有些情况下会带来很大的不便,那么应该怎末解决呢?

(1) 在资源编辑器中创建工具栏,由于它只是作为今后处理的原始材料,因此其中按钮图案是什么并不重要。

(2) 在其他图像编辑器中(例如画笔)创建工具栏位图。位图尺寸必须与资源编辑器中创建的工具栏一致。

(3) 将新创建的位图导入应用程序资源中,并将其 ID 设置为 IDB_TOOLBARHI。

(4) 在 CMainFrame 类中声明一个 CBitmap 类型的变量 m_bmToolbarHi。

(5) 在 CMainFrame::OnCreate 函数中定制创建工具栏的代码,如下所示:

```
if (! m_wndToolBar.Create(this) || ! m_wndToolBar.LoadToolBar(IDR_MAINFRAME))  
{  
    TRACE0("Failed to create toolbar\n");  
    return -1; // fail to create  
}
```

```
m_bmToolbarHi.LoadBitmap( IDB_TOOLBARHI );  
m_wndToolBar.SetBitmap( (HBITMAP)m_bmToolbarHi );
```

(6) 在 CMainFrame 类的析构函数中添加如下代码:

```
m_bmToolbarHi.DeleteObject();
```

8.7 去除浮动工具栏中的系统菜单

工具栏一般停靠在窗口边缘,当使用鼠标将其拖动到窗口客户区中(或双击它),它就会变成如图 8-6 所示的浮动状态。此时 MFC 框架会自动为浮动工具栏添加系统菜单。那么是否能够去除系统菜单呢?实际上,只要使工具栏对象在鼠标双击或拖动时,设法得到系统菜单的句柄并将其去除即可。下面是具体的实现步骤:

(1) 创建 CToolBar 的派生类 CMyToolBar,用以完成去除系统菜单的操作。

(2) 在 CMyToolBar 类中处理鼠标双击消息 WM_LBUTTONDOWNBLCK。

在该消息的处理函数 OnLButtonDownBlk 中,先进
行 CToolBar 的默认处理。然后判断工具栏是否处于
浮动状态,如果是则使用 GetParent 函数得到其父窗口句柄,并由此句柄调用 ModifyStyle 函



图 8-6 浮动工具栏

数去除系统菜单。清单 8-5 所示为 OnLButtonDblClk 函数的源代码：

清单 8-5 OnLButtonDblClk() 函数

```
CMyToolBar::OnLButtonDblClk(nFlags, point)
{
    CToolBar::OnLButtonDblClk(nFlags, point);
    if (IsFloating()) //工具栏是否浮动
    {
        CWnd* pMiniFrame;
        CWnd* pDockBar;

        pDockBar = GetParent();
        pMiniFrame = pDockBar->GetParent();
        //除去系统菜单
        pMiniFrame->ModifyStyle(WS_SYSMENU, NULL);
        //重新绘制窗口
        pMiniFrame->ShowWindow(SW_HIDE);
        pMiniFrame->ShowWindow(SW_SHOW);
    }
}
```

需要注意的是,在函数的末尾连续调用 ShowWindow 函数,其目的就在于强制浮动窗口进行重新绘制,以体现出窗口的变化。

(3) 响应鼠标左键单击消息 WM_LBUTTONDOWN,并在其消息处理函数中进行同样的处理。

(4) 将 CMainFrame 中的 m_wndToolBar 成员(或其他想去除系统菜单的工具栏对象)类型修改为 CMyToolBar 即可。

8.8 排列多个工具栏

当应用程序中用到多个工具栏时,如果都使其具有顶端停靠的风格,则默认情况下这些工具栏将按照创建顺序,由上到下的排列。如果每个工具栏中的按钮都比较多,所有工具栏排在一行比较困难,那么这种排列方式倒也没有太大影响。然而如果有的工具栏中只包括很少的按钮,那么这样排列就会浪费宝贵的屏幕空间(当然,用户可以手动将其拖放到一行显示)。而如果要将其其中的一些合并,又不见得合适。例如,当用户选择了屏幕中的某个对象后,出现的针对该对象的工具栏就不能随意与其他工具栏合并。

那么最合适的解决方法只有一种,定制代码尽可能将工具栏从左到右顺序排列,而尽量不出现一个工具栏占据一行的情况。图 8-7 即为两个工具栏按左右顺序排列的情况。

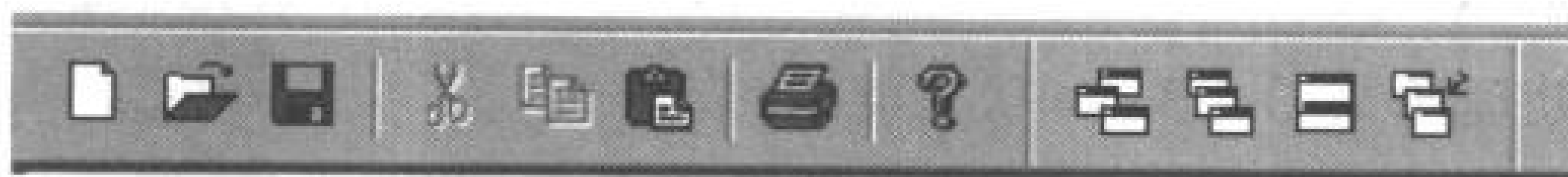


图 8-7 顺序排列的工具栏

首先在 CMainFrame 类中(或者在其他将显示工具栏的类)定制停靠函数: DockControlBarLeftOf, 其源代码如清单 8-6 所示, 其中 Bar 指定了行中右侧的工具栏; LeftOf 则指定了行中左侧的工具栏。在函数中根据左侧工具栏的位置和停靠方式, 决定右侧工具栏的停靠位置和停靠方式。

清单 8-6 DockControlBarLeftOf() 函数

```
void CMainFrame::DockControlBarLeftOf(CToolBar * Bar, CToolBar * LeftOf)
{
    CRect rect;
    DWORD dw;
    UINT n;

    // 使 MFC 调整所有停靠工具栏的尺寸, 这样 GetWindowRect 将得到正确的结果
    RecalcLayout(TRUE);
    LeftOf->GetWindowRect(&rect);
    rect.OffsetRect(1,0);
    dw = LeftOf->GetBarStyle();
    n = 0;
    n = (dw&CBRS_ALIGN_TOP) ? AFX_IDW_DOCKBAR_TOP : n;
    n = (dw&CBRS_ALIGN_BOTTOM && n == 0) ? AFX_IDW_DOCKBAR_BOTTOM : n;
    n = (dw&CBRS_ALIGN_LEFT && n == 0) ? AFX_IDW_DOCKBAR_LEFT : n;
    n = (dw&CBRS_ALIGN_RIGHT && n == 0) ? AFX_IDW_DOCKBAR_RIGHT : n;

    // 当使用默认参数时, DockControlBar 会将每个工具栏停靠在单独的一行
    // 通过计算矩形, 能够使工具栏被拖动并停靠到合适的位置
    DockControlBar(Bar,n,&rect);
}
```

此时, 在 CMainFrame::OnCreate 函数中使用 DockControlBarLeftOf, 而不是 DockControlBar 来设置控件的停靠, 示范代码如下:

```
m_wndToolBar1.EnableDocking(CBRS_ALIGN_ANY);
m_wndToolBar2.EnableDocking(CBRS_ALIGN_ANY);
EnableDocking(CBRS_ALIGN_ANY);
DockControlBar(&m_wndToolBar1);
DockControlBarLeftOf(&m_wndToolBar2,&m_wndToolBar1);
```

其中 m_wndToolBar2 和 m_wndToolBar1 分别为两个工具栏对象, 代码将使 m_wndToolBar2 排列在 m_wndToolBar1 左边。

8.9 在对话框中使用工具栏和工具提示

对话框中主要出现的是按钮、编辑框等控件, 而不是工具栏。但是在有些情况下, 使用工具栏能够简化操作, 并为用户提供更多的交互方式。

8.9.1 创建工具栏

在对话框中创建工具栏与在框架窗口中创建工具栏方法一样,首先载入工具栏资源,然后设置按钮 ID 和风格,具体操作如下所示:

```
// 创建工具栏
if (m_toolBar.Create(this))
{
    //载入用于创建工具栏的位图
    m_toolBar.LoadBitmap(IDB_TOOLBAR);
    //设置工具栏按钮
    m_toolBar.SetButtons(auIDToolBar, sizeof(auIDToolBar)/sizeof(UINT));
}

//设置工具栏风格
m_toolBar.GetToolBarCtrl().SetStyle(TBSTYLE_FLAT | CBRs_ALIGN_TOP | CBRs_
    TOOLTIPS | CBRs_FLYBY | CBRs_SIZE_FIXED);
```

8.9.2 修改对话框尺寸

但是情况并不那么简单,如果现在运行程序,就会发现工具栏并不一定会出现在对话框窗口中。这是因为在对话框中创建工具栏时,工具栏所占的空间也属于对话框窗口客户区(并不象在框架窗口中那样,占据的是非客户区),因此如果对话框中没有足够的空间的话,工具栏就可能会被对话框中的其他控件所覆盖。所以在创建工具栏时,必须保证足够的空间来放置它们。

一般使用的方法就是在创建工具栏时,先计算出它的尺寸,再将对话框的客户区扩大到相应尺寸,这样就不会出现被覆盖的情况。具体实现代码如下所示:

```
//确定工具栏的大小后,重新设置对话框大小以放置工具栏
CRect rcClientStart;// 原来的窗口大小
CRect rcClientNow;// 放置工具栏后窗口的大小
GetClientRect(rcClientStart);
RepositionBars(AFX_IDW_CONTROLBAR_FIRST, AFX_IDW_CONTROLBAR_LAST, 0, repos-
    Query, rcClientNow);

// 移动对话框中的所有控件使之不会被工具栏覆盖
CPoint ptOffset(rcClientNow.left - rcClientStart.left, rcClientNow.top - rc-
    ClientStart.top);

CRect rcChild;
CWnd* pwndChild = GetWindow(GW_CHILD); //得到第一个控件的指针
while (pwndChild)
{
    pwndChild->GetWindowRect(rcChild); //得到控件矩形
    ScreenToClient(rcChild); //坐标变换
```



```

    rcChild.OffsetRect(ptOffset);    //移动控件
    pwndChild->MoveWindow(rcChild, FALSE);
    pwndChild = pwndChild->GetNextWindow();    //得到下一个控件的指针
}

// 调整对话框尺寸
CRect rcWindow;
GetWindowRect(rcWindow);
rcWindow.right += rcClientStart.Width() - rcClientNow.Width();
rcWindow.bottom += rcClientStart.Height() - rcClientNow.Height();
MoveWindow(rcWindow, FALSE);

// 将工具栏定位
RepositionBars(AFX_IDW_CONTROLBAR_FIRST, AFX_IDW_CONTROLBAR_LAST, 0);

```

8.9.3 显示工具提示

在显示工具提示前,TTN_NEEDTEXT 消息会发送给工具栏的父窗口,以获得将要显示的工具提示。在框架窗口中显示工具提示不需要用户作什么额外工作,这是因为所有由 CFrameWnd 派生的框架窗口都能够默认处理 TTN_NEEDTEXT 消息。当然,用户可以通过设置工具栏风格之类的方法来禁止或允许工具栏提示,但仅此而已。

然而如果框架窗口不是由 CFrameWnd 类派生而来的,如从对话框或格式视(Form View)派生,那么用户就必须自行添加处理 TTN_NEEDTEXT 消息的消息映射,以显示工具提示。下面的语句为为消息映射提供的入口:

```
ON_NOTIFY_EX( TTN_NEEDTEXT, 0, memberFxn );
```

其中 TTN_NEEDTEXT 为所处理的消息,0 为工具提示的 ID(工具提示的 ID 总是 0),memberFxn 为处理消息的函数。

具体在编程时,首先在头文件中添加对消息处理函数的声明:

```

//{{AFX_MSG(CMediaPlayerDlg)
...
//{{AFX_MSG
afx_msg BOOL OnToolTipNotify( UINT id, NMHDR * pNMHDR, LRESULT * pResult );

```

注意 函数的声明前一定要加 afx_msg,并且必须位于消息处理函数群外。

然后向源文件中加入消息映射入口:

```

BEGIN_MESSAGE_MAP(CMediaPlayerDlg, CDialog)
    //{{AFX_MSG_MAP(CMediaPlayerDlg)
    ...
    //{{AFX_MSG_MAP
    ON_NOTIFY_EX(TTN_NEEDTEXT,0,OnToolTipNotify)
END_MESSAGE_MAP()

```

注意 消息映射入口必须添加在消息映射中。

清单 8-7 为消息处理函数的源代码：

清单 8-7 OnToolTipNotify() 函数

```

BOOL CMediaPlayerDlg::OnToolTipNotify( UINT id, NMHDR * pNMHDR, LRESULT * pResult )
{
    TOOLTIPTEXT * pTTT = (TOOLTIPTEXT *)pNMHDR;    //取得工具提示结构数据
    UINT nID = pNMHDR->idFrom;    // idFrom 为工具栏的句柄
    // 使框架处理消息
    if (GetRoutingFrame() != NULL)
        return FALSE;

    TCHAR szFullText[256];
    CString strTipText;
    //如果为 TTN_NEEDTEXT 消息才进行处理
    if (pNMHDR->code == TTN_NEEDTEXT && (pTTT->uFlags & TTF_IDISHWND))
    {
        nID = ((UINT)(WORD)::GetDlgCtrlID((HWND)nID));
    }

    if(nID != 0) // 如果不是工具栏间隔才有工具栏提示
    {
        AfxLoadString(nID, szFullText);    //载入相应的工具条提示
        AfxExtractSubString(strTipText, szFullText, 1, '\n');    //截取\n后面的文本
    }

    //将工具条提示文本拷入将用于显示工具条提示的 String 内
    lstrcpyn(pTTT->szText, strTipText,
        (sizeof(pTTT->szText)/sizeof(pTTT->szText[0])));

    *pResult = 0;

    // 使工具提示窗口显示于所有窗口之上
    ::SetWindowPos(pNMHDR->hwndFrom, HWND_TOP, 0, 0, 0, 0,
        SWP_NOACTIVATE|SWP_NOSIZE|SWP_NOMOVE);

    return TRUE;    // 消息处理成功
}

```

读者现在运行应用程序,将鼠标滑过工具栏按钮,看看会发生什么。出乎意料,什么也没有发生,工具提示并没有如预期地那样出现。这是因为用户在应用程序中还没有允许工具提示,换句话说就是工具提示是空的,没有信息可以显示。如果读者在该消息处理函数中加入消息框输出语句,如:

```
AfxMessageBox("消息处理了吗?");
```

就会发现当鼠标掠过工具栏时,会出现消息框。也就是说消息处理函数正常运行。那么唯一的可能就是工具提示没有被允许,因此向 OnInitialUpdate() 函数中加入下面的语句:

```
m_toolBar.EnableToolTips(TRUE);
```

现在将正常显示工具提示。

8.10 在 MDI 应用程序中切换工具栏

对于 MDI 应用程序来说,其中不同的子窗口往往具有不同的功能,因而使用的工具栏按钮也不尽相同。然而在默认情况下, `CMainFrame::OnCreate` 创建的工具栏不能被动态修改。也就是说如果不使用定制的代码,只能将所有的工具栏都显示出来,这显然无助于改善用户界面。本节将向读者介绍如何在窗口中切换工具栏,当然主框架工具栏将总是被显示的,例如“保存”、“新建”等(虽然对于不同的窗口,可能保存或新建的内容也会不同)。

要在不同的窗口间切换不同的工具栏,就需要从窗口中得到与其相联系的工具栏 ID。为此,首先确定活动的 MDI 子窗口,并由之得到其文档类。然后再通过文档类得到其文档模板类,而工具栏 ID 正是由该类负责维护的。在 MDI 中每切换到一个新窗口时,就响应其客户窗口(`CMainClient`)发送的 `WM_MDISETMENU` 消息,并执行上述操作。

我们在主框架窗口中声明两个数组:一个用于保存已经载入的工具栏指针,另一个用于保存已经载入的工具栏 ID。当设置了一个新菜单时,主框架会在工具栏 ID 数组中进行检索。如果找到匹配项则将其显示在屏幕上;否则将创建新的工具栏,并将工具栏指针和 ID 存放在对应数组中。

下面为实现上述想法的步骤:

- (1) 为每个文档/视图类创建工具栏,并将其 ID 就设置为文档所使用的菜单 ID。
- (2) 创建 `CWnd` 的派生类 `CMainClient`,用于管理 MDI 的客户区。

在该类中添加两个消息处理函数,清单 8-8 所示为 `DefWindowProc` 函数的源代码。该函数由主框架窗口调用,在设置新的 MDI 菜单时负责切换工具栏。

清单 8-8 `DefWindowProc()` 函数

```
LRESULT CMainClient::DefWindowProc(UINT message, WPARAM wParam, LPARAM lParam)
{
    LRESULT lRet = CWnd::DefWindowProc(message, wParam, lParam);
    if (message == WM_MDISETMENU) {
        CMainFrame *pFrame;

        pFrame = (CMainFrame *) AfxGetMainWnd();
        if (::IsWindow(pFrame))
            pFrame->SwitchToolbar();
    }
    return (lRet);
}
```

清单 8-9 所示为 `OnParentNotify` 函数,主框架窗口将在 MDI 子窗口被关闭时,调用它来切换工具栏。

清单 8-9 `OnParentNotify()` 函数

```
void CMainClient::OnParentNotify(UINT message, LPARAM lParam)
{
    ...
}
```

```

CWnd::OnParentNotify(message, lParam);
CMainFrame * pFrame;
CWnd * pWnd;
pWnd = AfxGetMainWnd();
if (pWnd->IsKindOf(RUNTIME_CLASS(CMainFrame)))
    pFrame = (CMainFrame *) AfxGetMainWnd();
else
    return;
switch (LOWORD(message)) {
case WM_CREATE:
    m_nChilds++;
    break;
case WM_DESTROY:
    m_nChilds--;
    if (m_nChilds == 0)
        pFrame->SwitchToolbar();
    break;
default:
    break;
}
}

```

(3) 创建 CDocTemplate 的派生类 CDocTemplateEx, 并在该类中添加 GetResourceID 函数, 其源代码如清单 8-10 所示:

清单 8-10 GetResourceID() 函数

```

int CDocTemplateEx::GetResourceID()
{
    return (m_nIDResource);
}

```

(4) 在 CMainFrame 类中添加下列成员变量:

```

CArray m_aToolbars;
CArray m_idrLoaded;
CMainClient m_wndClient;

```

其中 m_aToolbars 中包含了工具栏指针, m_idrLoaded 中包含了工具栏 ID, m_wndClient 则为 CMainClient 对象。

(5) 在 CMainFrame::OnCreate 函数中添加如下代码(黑体文字):

```

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CMDIFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;
    m_wndClient.SubclassWindow(m_hWndMDIClient);
    ...
}

```

此行代码将 MDI 客户区对象归类为 CMainClient, 也就是由该类来管理应用程序的客

户区。这样就能够利用 CMainClient 类中的附加功能了。

(6) 在 CMainFrame 类中设计用于切换、设置工具栏的函数。

SwitchToolbar 函数用于从当前视图中得到资源 ID, 该函数接着调用 SetToolbar 函数以激活指定的工具栏, 并解除其他工具栏的激活状态。如果当前没有视图打开资源 ID, 则将相应的 idResource 设置为 0, 并解除所有工具栏的激活状态。清单 8-11 所示为 SwitchToolbar 函数的源代码:

清单 8-11 SwitchToolbar() 函数

```
void CMainFrame::SwitchToolbar ()
{
    CView * pView;
    CDocTemplateEx * pTmpl;
    CMDIChildWnd * pKid;
    int idResource = 0;

    pKid = MDIGetActive ();
    if (::IsWindow (pKid)) {
        pView = pKid->GetActiveView ();
        if (::IsWindow (pView)) {
            pTmpl = (CDocTemplateEx *) pView->GetDocument ()->GetDocTemplate ();
            idResource = pTmpl->GetResourceID ();
        }
    }

    SetToolbar (idResource);
}
```

SetToolbar 函数用于从工具栏数组中得到工具栏索引: m_vToolbars。这将通过在数组中循环检索完成。SetToolbar 函数的源代码如清单 8-12 所示:

清单 8-12 SetToolbar() 函数

```
void CMainFrame::SetToolbar (int nIdr)
{
    static int nToolbars;
    int n, nIdx;
    CString str;
    CToolBar * pTB;

    str.Format ("Toolbar # %d", ++ nToolbars);
    nIdx = AddToolbar (nIdr, str);
    for (n=0 ; n < m_vToolbars.GetSize () ; n++) {
        pTB = m_vToolbars.GetAt (n);
        if (n != nIdx) {
            ShowControlBar (pTB, 0, 0);
        }
        else
            ShowControlBar (pTB, 1, 1);
    }
    m_wndToolBar.GetParentFrame ()->RecalcLayout ();
}
```


AddToolBar 函数用于获取指定 ID(nIDR)的工具栏索引。如果在 ID 数组中不存在指定 ID,则函数将创建一个新的工具栏,并将其指针和 ID 添加到相应数组中。AddToolBar 函数的源代码如清单 8-13 所示:

清单 8-13 AddToolBar()函数

```
int CMainFrame::AddToolBar (int nIDR, CString strWndTxt)
{
    CToolBar * pTB;
    BOOL f;
    CRect rc;
    int nIdx;

    if (nIDR == 0)
        return (-1);
    if ((nIdx = ::FindInArray (m_idrLoaded, nIDR)) < 0) {
        pTB = new CToolBar;
        pTB->Create (this);
        f = pTB->LoadToolBar (nIDR);
        if (f == 0) {
            pTB->DestroyWindow ();
            delete pTB;
            return (-1);
        }
        pTB->SetBarStyle(m_wndToolBar.GetBarStyle() |
            CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);
        pTB->EnableDocking(CBRS_ALIGN_TOP);
        m_wndToolBar.GetWindowRect (&rc);
        rc.OffsetRect(1,0);
        DockControlBar(pTB, AFX_IDW_DOCKBAR_TOP, &rc);
        pTB->SetWindowText (strWndTxt);
        pTB->ShowWindow (SW_RESTORE);
        ShowControlBar (pTB, 1, 1);
        nIdx = m_vToolbars.Add (pTB);
        m_idrLoaded.Add (nIDR);
    }
    return (nIdx);
}
```

本章小结

本章主要介绍的是工具栏编程的基本思路和方法。通过本章的学习,读者应该达到以下几点:

- 掌握 CToolBar 类的使用。
- 掌握定制工具栏的方法和思路。

第9章 状态栏

状态栏用于给出当前操作的提示,它与菜单和工具栏一起组成了 Windows 应用程序的标准界面。本章将向读者介绍定制工具栏的方法。

本章要点:

- CStatusBar 类的使用;
- 标准状态栏的使用;
- 在状态栏中实现滚动文本效果;
- 在状态栏中输出时间;
- 在状态栏中使用控件。

9.1 状态栏编程基础

形象地说 CStatusBar 对象类似于包括一行文本输出栏(窗格)的控件栏。它们通常用来显示当前的状态信息,例如显示如何使用当前菜单命令的帮助信息。状态栏是由 CStatusBar 类进行管理的,图 9-1 显示了 CStatusBar 类的派生结构图。

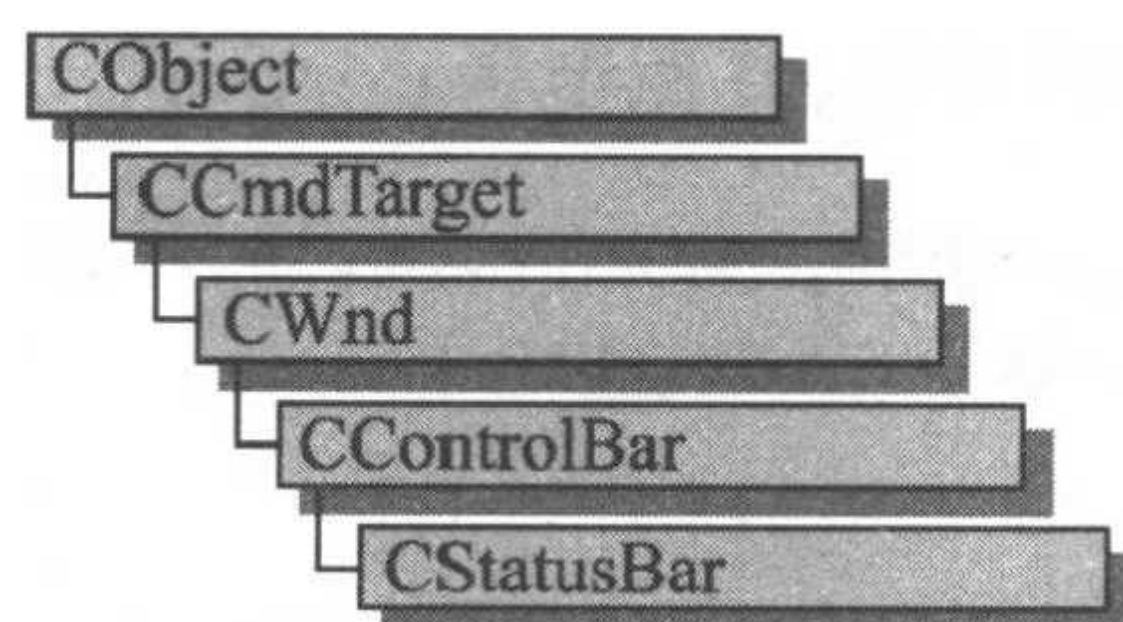


图 9-1 CStatusBar 类的派生结构图

9.1.1 状态栏概述

MFC 4.0 新引入的 CToolBar::GetStatusBarCtrl 函数提供了获得 Windows 为状态栏添加的扩展功能的手段。

框架窗口将窗格信息储存在一个数组中,并将最左边的窗格的索引定义为 0。当用户创建状态栏时,使用的是一个 ID 数组,框架将自动将其与相应的窗格相联系。

默认情况下,第一个窗格(0)是具有“弹性”的,也就是说它将占据状态栏中所有空余长度,亦即状态栏窗格是右对齐的。

创建状态栏的一般步骤如下:

- (1) 创建 CStatusBar 对象。
- (2) 调用 Create 或 CreateEx 函数以创建状态栏窗口,并将其与 CStatusBar 对象相联系。
- (3) 调用 SetIndicators 函数为每个窗格设置 ID。

有三种方法可以更新状态栏中的文本:

- (1) 调用 CWnd::SetWindowText 函数,但这种方法只能更新窗格 0 中的文本。
- (2) 在状态栏窗格的 ON_UPDATE_COMMAND_UI 消息处理函数中,调用 CCmdUI::

SetText 函数。

- (3) 调用 SetPaneText 函数可以更新任何窗格中的文本。

调用 SetPaneStyle 可以更新状态栏窗格的风格。

在 CStatusBar 类中封装了用于创建、设置状态栏的成员函数,类的常用成员函数如下所示。

9.1.2 构造函数

CStatusBar 类的构造函数包括: CStatusBar、Create、CreateEx 和 SetIndicators,它们可以完成创建状态栏对象、设置状态栏窗格 ID 等操作。

- CStatusBar

调用该函数以构造状态栏对象,其原型为:

```
CStatusBar( );
```

该函数将以默认字体创建状态栏对象。

- Create

调用该函数可以创建状态栏对象,其原型为:

```
BOOL Create( CWnd* pParentWnd, DWORD dwStyle = WS_CHILD | WS_VISIBLE | CBRSBOTTOM, UINT nID = AFX_IDW_STATUS_BAR );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

pParentWnd ——指定了状态栏的主框架窗口。

dwStyle ——指定了状态栏的风格,其风格常量除了可以为 Windows 常规风格外,还可取的附加风格常量如表 9-1 所示。

表 9-1 状态栏附加风格常量

附加风格常量	含义
CBRS_TOP	状态栏在框架窗口的顶部
CBRS_BOTTOM	状态栏在框架窗口的底部
CBRS_NOALIGN	框架窗口改变尺寸时,状态栏不重新定位

nID ——指定了状态栏主窗口 ID。

- CreateEx

调用该函数可以创建具有扩展风格的状态栏对象,其原型为:

```
BOOL CreateEx( CWnd* pParentWnd, DWORD dwCtrlStyle = 0, DWORD dwStyle = WS_CHILD | WS_VISIBLE | CBRs_BOTTOM, UINT nID = AFX_IDW_STATUS_BAR );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

pParentWnd ——指定了状态栏的主框架窗口。

dwCtrlStyle ——指定了状态栏扩展风格,如表 9-2 所示。

表 9-2 状态栏扩展风格常量

附加风格常量	含义
SBARS_SIZEGRIP	在状态栏最右端有尺寸调整把手
SBT_TOOLTIPS	状态栏支持工具提示

dwStyle ——指定了状态栏的风格,其风格常量如表 9-2 所示。

nID ——指定了状态栏主窗口 ID。

- SetIndicators

调用该函数可以设置各个指示窗格的 ID,其原型为:

```
BOOL SetIndicators( const UINT* lpIDArray, int nIDCount );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

lpIDArray ——为包含状态栏指示窗格 ID 的数组。

nIDCount ——指定了状态栏指示窗格数目。

SetIndicators 函数使用 lpIDArray 数组中的相应元素,设置状态栏对应窗格的 ID;载入由 ID 指定的字符串资源,并根据其设置状态栏窗格文本。

9.1.3 属性操作函数

CStatusBar 类的属性操作函数包括: CommandToIndex、GetItemID、GetItemRect、GetPaneInfo、GetPaneStyle、GetPaneText、GetStatusBarCtrl、GetPaneStyle、SetPaneText 和 SetPaneInfo,它们可以完成创建状态栏对象、设置状态分栏 ID 等操作。

- CommandToIndex

调用该函数可以获得给定 ID 值的状态栏指示窗格索引,其原型为:

```
int CommandToIndex( UINT nIDFind ) const;
```

返回值:

如果函数调用成功,则返回指示窗格索引,否则返回 -1。

参数:

nIDFind ——指定了将获取其索引的窗格 ID。

- GetItemID

调用该函数可以获得给定索引的状态栏指示窗格的 ID,其原型为:

```
UINT GetItemID( int nIndex ) const;
```

返回值:

如果函数调用成功,则返回指示窗格的 ID。

参数:

nIndex ——指定了将获取其 ID 的状态栏窗格索引。

- GetItemRect

调用该函数可以获得给定指示窗格的客户区矩形,其原型为:

```
void GetItemRect( int nIndex, LPRECT lpRect ) const;
```

参数:

nIndex ——指定了将获取其客户区矩形的状态栏指示窗格索引。

lpRect ——将返回状态栏指示窗格的尺寸矩形。

GetItemRect 函数将相应状态栏指示窗格的坐标,拷贝到 lpRect 结构中。坐标是以像素为单位的,并且其值是相对于状态栏左上角的。

- GetPaneInfo

调用该函数可以获得给定指示窗格的 ID、风格和宽度,其原型为:

```
void GetPaneInfo( int nIndex, UINT& nID, UINT& nStyle, int& cxWidth ) const;
```

参数:

nIndex ——指定了将获取其信息的状态栏窗格索引。

nID ——将返回状态栏指示窗格的 ID。

nStyle ——将返回状态栏指示窗格的风格。

cxWidth ——将返回状态栏指示窗格的宽度。

- GetPaneStyle

调用该函数可以获得给定指示窗格的风格,其原型为:

```
UINT GetPaneStyle( int nIndex ) const;
```

返回值:

函数调用成功,则返回状态栏指示窗格的风格。

参数:

nIndex ——指定了将获取其风格的状态栏指示窗格索引。

- GetPaneText

调用该函数可以获得给定指示窗格的文本,其原型为:


```
CString GetPaneText( int nIndex ) const;  
void GetPaneText( int nIndex, CString& rString ) const;
```

返回值:

如果函数调用成功,则返回包含指示窗格文本的 CString 对象。

参数:

nIndex ——指定了将获取其风格的状态栏指示窗格索引。

rString ——将返回状态栏指示窗格文本。

- GetStatusBarCtrl

调用该函数可以获得状态栏控制对象指针,其原型为:

```
CStatusBarCtrl& GetStatusBarCtrl( ) const;
```

返回值:

如果函数调用成功,则返回 CStatusBarCtrl 对象指针。

获得状态栏控制对象的指针后,可以使用该对象的高级函数来对状态栏进行管理。

- SetPaneInfo

调用该函数可以设置给定指示窗格的 ID、风格和宽度,其原型为:

```
void SetPaneInfo( int nIndex, UINT nID, UINT nStyle, int cxWidth );
```

参数:

nIndex ——指定了将设置其风格的状态栏指示窗格索引。

nID ——指定了将设置的指示窗格 ID。

nStyle ——指定了将设置的指示窗格风格。

cxWidth ——指定了将设置的指示窗格宽度。

- SetPaneStyle

调用该函数可以设置给定指示窗格的风格,其原型为:

```
void SetPaneStyle( int nIndex, UINT nStyle );
```

参数:

nIndex ——指定了将设置其风格的状态栏指示窗格索引。

nStyle ——指定了将设置的指示窗格风格。

- SetPaneText

调用该函数可以设置给定指示窗格的文本,其原型为:

```
BOOL SetPaneText( int nIndex, LPCTSTR lpszNewText, BOOL bUpdate = TRUE );
```

返回值:

如果函数调用成功,则返回非零值,否则返回零值。

参数:

nIndex ——指定了将设置其文本的状态栏指示窗格索引。

lpszNewText ——指定了将设置的指示窗格文本。

bUpdate ——如果该参数为 TRUE,则指示窗格的显示将被更新。

9.1.4 重载函数

CStatusBar 类的重载函数为 DrawItem,它用于在自绘制状态栏控件的可见区域发生变化时进行重绘,其原型为:

```
virtual void DrawItem( LPDRAWITEMSTRUCT lpDrawItemStruct );
```

参数:

lpDrawItemStruct ——指向包含所需绘制类型信息的 DRAWITEMSTRUCT 结构。

9.2 使用标准状态栏

应用程序一般通过重载 CMainFrame 类中的 OnCreate 函数载入状态栏,下面以在应用程序中使用状态栏的实现代码为例向读者介绍使用状态栏的一般步骤:

(1) 定义状态栏各窗格字符串 ID 数组

```
static UINT indicators[] =
{
    ID_SEPARATOR, // 状态栏第一窗格,提示信息输出栏
    ID_SEPARATOR, // 状态栏第二窗格,显示鼠标位置
    ID_INDICATOR_CAPS, // 状态栏第三窗格,显示 Caps Lock 键状态
    ID_INDICATOR_NUM, // 状态栏第四窗格,显示 Num Lock 键状态
    ID_INDICATOR_SCRL, // 状态栏第五窗格,显示 Scroll Lock 键状态
};
```

(2) 声明状态栏对象

```
class CMainFrame : public CMDIFrameWnd
{
...
protected: // control bar embedded members
    CStatusBar m_wndStatusBar;
...
};
```

(3) 创建状态栏

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
...
//载入状态栏
if (! m_wndStatusBar.Create(this) ||
    ! m_wndStatusBar.SetIndicators(indicators,
    sizeof(indicators)/sizeof(UINT)))
```

```

    {
        TRACE0("Failed to create status bar\n");
        return -1; // fail to create
    }
    //设置状态栏第二窗格,鼠标位置的初始显示文本
    m_wndStatusBar.SetPaneText( 1, "x=10,y=10",TRUE );
    m_wndStatusBar.SetPaneInfo( 1, ID_SEPARATOR,SBPS_STRETCH, 10);
    ...
}

```

创建状态栏时,在鼠标位置窗格中显示的坐标为静态值,在以后的程序中将在该窗格中动态显示鼠标坐标。具体来说就是响应鼠标移动消息 WM_MOUSEMOVE,根据鼠标的不同位置调用 SetPaneText 函数设置窗格文本即可。

9.3 在状态栏中显示滚动效果的文本

滚动效果的文本经常可以在应用程序中看到,例如在“关于”对话框中显示致谢信息(垂直滚动),或在闪屏效果中显示欢迎文本(水平滚动)。在本节中,我们将在状态栏中实现水平滚动文本的效果。具体步骤如下:

(1) 创建 CStatusBar 的派生类 CMyStatusBar,用于管理显示滚动效果文本的状态栏对象。

(2) 在 CMainFrame 类(或其他创建状态栏的类)中将状态栏对象类型修改为 CMyStatusBar。

```
CMyStatusBar m_wndStatusBar;
```

(3) 定义状态栏各窗格字符串 ID 数组如下:

```

static UINT indicators[] =
{
    ID_SEPARATOR,
    IDS_SCROLL_PANE, //滚动文本窗格
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
};

```

(4) 在字符串表资源中添加 IDS_SCROLL_PANE,并将其内容输入一些空格。状态栏中滚动文本窗格的尺寸将由这些空格决定。当然也有其他设置窗格尺寸的方法,但是这无疑是最直观的一种。

(5) 在 CMyStatusBar 类中添加一个 CString 型变量 m_strScrollText,用于表示将在其中显示的文本信息。

(6) 在 CMyStatusBar 类中处理定时器消息。

由于我们的目的是在状态栏中显示滚动文本,也就是说文本从窗格左边出现,再滚动到窗格右边消失,并周而复始地重复这个过程。这个效果实现方法很简单:开始时只显示一个字符,然后每隔一段时间增加一个字符即可。因此需要处理定时器消息,清单 9-2 所示为 OnTimer 函数的源代码:

清单 9-2 OnTimer() 函数

```
void CMyStatusBar::OnTimer(UINT nIDEvent)
{
    if (m_strScrollText.IsEmpty())
    {
        KillTimer(1);
        SetPaneText(CommandToIndex(IDS_SCROLL_PANE), "");
        return;
    }
    static UINT str_idx = 0; //字符串的偏移

    // 如果已经到达字符串的末尾,则返回开始
    if (str_idx >= (UINT)(m_strScrollText.GetLength() / 2) - 1)
    {
        str_idx = 0;
    }

    //显示字符串
    SetPaneText(CommandToIndex(IDS_SCROLL_PANE), ((LPCSTR)
        m_strScrollText) + str_idx);

    //滚动一个字符
    str_idx = str_idx + 1;

    CStatusBar::OnTimer(nIDEvent);
}
```

(7) 销毁定时器。

由于水平滚动效果将在状态栏的生命期内始终显示,因此在状态栏被销毁时也应该将定时器对象销毁。清单 9-3 所示为 OnDestroy 函数的源代码:

清单 9-3 OnDestroy() 函数

```
void CMyStatusBar::OnDestroy()
{
    CStatusBar::OnDestroy();
    KillTimer(1);
}
```

(8) 在类中添加用以开始滚动文本的函数 StartDisplay。

该函数应该在框架(状态栏)被创建后被调用,一般应该在 CWinApp::InitInstance 函数中进行。清单 9-4 所示为 StartDisplay 函数的源代码:

清单 9-4 StartDisplay() 函数

```
void CMyStatusBar::StartDisplay(void)
```

```

{
    //设置滚动的文本
    m_strScrollText = "你好,欢迎使用!"
    m_strScrollText += m_strScrollText;
    KillTimer(1);
    VERIFY(SetTimer(1, 200, NULL) != 0);
}

```

函数将滚动文本进行了硬编码,读者可以为 StartDisplay 设置参数,以从外部设置滚动文本。

9.4 在状态栏中输出时间

在应用程序的状态栏中添加时间显示是一个很好的辅助功能,不但能够使用户很方便地得到时间信息,而且有时也能作为应用程序功能的一部分(例如,在播放器中显示媒体当前播放进度等)。具体实现步骤如下:

(1) 创建 CStatusBar 的派生类 CMyStatusBar,用于管理显示时钟的状态栏对象。

CMyStatusBar 类的定义如清单 9-5 所示:

清单 9-5 CMyStatusBar 类定义

```

class CMyStatusBar : public CStatusBar {
    DECLARE_DYNCREATE(CMyStatusBar)
public:
    CMyStatusBar();
    ~CMyStatusBar();
private:
    CString m_strClockFormat;
public:
    void SetClockFormat(LPCTSTR strClockFormat);

    //{AFX_MSG(CMyStatusBar)
    afx_msg void OnDestroy();
    afx_msg void OnUpdateIndicatorTime(CCmdUI * pCmdUI);
    afx_msg int OnCreate( LPCREATESTRUCT lpCreateStruct );
    //{AFX_MSG
    DECLARE_MESSAGE_MAP()
};

```

其中成员变量 m_strClockFormat 为时间格式,用户可以通过设置该成员来改变状态栏中的时间显示格式。类的构造函数将为该成员设置默认的时间格式,清单 9-6 所示为其源代码:

清单 9-6 CMyStatusBar()函数

```

CMyStatusBar::CMyStatusBar()
: CStatusBar()

```



```
, m_strClockFormat("%H:%M:%S")
{
}
```

也可以使用 `SetClockFormat` 函数来显示地设置时间格式,其源代码如清单 9-7 所示:

清单 9-7 SetClockFormat() 函数

```
void CMyStatusBar::SetClockFormat(LPCTSTR strClockFormat)
{
    m_strClockFormat = strClockFormat;
}
```

在状态栏被创建时,设置定时器,并将其时间间隔设置为 1 秒。设置定时器的理由很简单,那就是时间必须每秒都被更新。清单 9-8 所示为 `OnCreate` 函数的源代码:

清单 9-8 OnCreate() 函数

```
int CMyStatusBar::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    // 当空闲时,每秒更新定时器一次
    CStatusBar::OnCreate(lpCreateStruct);
    SetTimer(ID_INDICATOR_TIME, 1000, NULL);
    return 0;
}
```

定时器消息的响应函数 `OnUpdateIndicatorTime` 负责更新时间显示,其源代码如清单 9-9 所示:

清单 9-9 OnUpdateIndicatorTime() 函数

```
void CMyStatusBar::OnUpdateIndicatorTime(CCmdUI * pCmdUI)
{
    pCmdUI->Enable(true);
    pCmdUI->SetText(CTime::GetCurrentTime().Format(m_strClockFormat));
}
```

由于时钟将在状态栏的生命期内始终显示,因此在状态栏被销毁时也应该将定时器对象销毁。清单 9-10 所示为 `OnDestroy` 函数的源代码:

清单 9-10 OnDestroy() 函数

```
void CMyStatusBar::OnDestroy()
{
    KillTimer(ID_INDICATOR_TIME);
    ProgressDestroy();
    CStatusBar::OnDestroy();
}
```

(2) 在字符串表资源中添加 `ID_INDICATOR_TIME`,并将其内容设置为 `HH:MM:SS`,以使窗格具有合适的空间。当用户改变时间格式时,必须首先确定空间是否足够,如果不够则应该改变窗格的尺寸。

(3) 定义状态栏各窗格字符串 ID 数组如下:

```
static const UINT indicators[] = {
    ID_SEPARATOR,
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
    ID_INDICATOR_TIME, // 时间窗格
};
```

(4) 在 CMainFrame 类(或其他创建状态栏的类)中将状态栏对象类型修改为 CMyStatusBar。

9.5 动态改变状态栏中的默认提示

Windows 应用程序的状态栏中显示的默认提示一般为“Ready”或“要获得帮助,请在帮助菜单中单击帮助主题”。本节将介绍如何动态修改默认提示。

与使用 `AFX_IDS_IDLEMESSAGE` 不同,这里将采用的技巧将是动态的。也就是当鼠标掠过工具栏按钮时,状态栏中出现对应提示时使用的机制,并且所出现的提示都是预先定义的静态文本。这一技术有很多用途,例如在处理数据库时,可以动态显示读入的记录数,或显示用于打开数据表所用的时间、数据表的名称等等。总之,它允许在应用程序运行时动态设置状态栏输出。

对应用程序的修改都是在 CMainFrame 类中进行。

(1) 在 CMainFrame 类中添加代表将在状态栏中输出的文本变量。

CString m_sStatusBarString;

(2) 在类中将创建两个函数：StatusBarMessage 和 OnSetMessageString。对这两个函数的声明如下所示：

```
//}{AFX_MSG(CMainFrame)
...
    afx_msg LRESULT OnSetMessageString(WPARAM wParam, LPARAM lParam);
...
//}}AFX_MSG
DECLARE_MESSAGE_MAP()

void StatusBarMessage(TCHAR * fmt,...);
```

(3) 在 MainFrm.cpp 中添加包含文件afxpriv.h, 其中包含了我们将使用的 WM_SETMESSAGESTRING 消息的定义。

(4) 在 CMainFrame 类中添加 WM_SETMESSAGESTRING 的消息映射。

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    ///{{AFX_MSG_MAP(CMainFrame)
    ...
```

```

ON_MESSAGE(WM_SETMESSAGESTRING, OnSetMessageString)
...
//{{AFX_MSG_MAP
END_MESSAGE_MAP()

```

(5) 设计 OnSetMessageString 函数。

OnSetMessageString 函数是对 CFrameWnd 类中相应函数的重载,其代码与 CFrameWnd::OnSetMessageString 函数几乎相同,唯一不同是显示定制的消息字符串。清单 9-11 所示为函数的源代码:

清单 9-11 OnSetMessageString() 函数

```

LRESULT CMainFrame::OnSetMessageString(WPARAM wParam, LPARAM lParam)
{
    UINT nIDLast = m_nIDLastMessage;
    m_nFlags &= ~WF_NOPOPMSG;

    CWnd* pMessageBar = GetMessageBar();
    if (pMessageBar != NULL)
    {
        CString sMsg;
        CString strMessage;

        // 设置文本
        if (lParam != 0)
        {
            ASSERT(wParam == 0);    // 不能得到 ID 和字符串
            m_sStatusBarString = (LPCTSTR)lParam;
            sMsg = m_sStatusBarString;
        }
        else if (wParam != 0)
        {
            // map SC_CLOSE to PREVIEW_CLOSE when in print preview mode
            if (wParam == AFX_IDS_SCCLOSE && m_lpfncloseProc != NULL)
                wParam = AFX_IDS_PREVIEW_CLOSE;

            // 得到由 wParam 指定的 ID 所对应的消息
            if (wParam == AFX_IDS_IDLEMESSAGE)
                sMsg = m_sStatusBarString;
            else
            {
                GetMessageString(wParam, strMessage);
                sMsg = strMessage;
            }
        }

        pMessageBar->SetWindowText(sMsg);

        // 使用最后被选择的消息更新状态栏
        CFrameWnd* pFrameWnd = pMessageBar->GetParentFrame();
        if (pFrameWnd != NULL)
        {

```

```

        m_nIDLastMessage = (UINT)wParam;
        m_nIDTracking = (UINT)wParam;
    }

    m_nIDLastMessage = (UINT)wParam;    // 新 ID (or 0)
    m_nIDTracking = (UINT)wParam;    // 使 F1 工作

    return nIDLast;
}

```

StatusBarMessage 函数主要由应用程序调用,以设置将在状态栏中显示的文本,然后触发 OnSetMessageString 函数,使文本显示于状态栏中。清单 9-12 所示为 StatusBarMessage 函数的源代码:

清单 9-12 StatusBarMessage() 函数

```

void CMainFrame::StatusBarMessage(TCHAR * fmt,...)
{
    TCHAR buffer[256] = _T("");
    CStatusBar* pStatus = (CStatusBar*)
        GetDescendantWindow(AFX_IDW_STATUS_BAR);
    va_list argptr;
    va_start(argptr, fmt);
    _vstprintf(buffer, fmt, argptr);
    va_end(argptr);
    m_sStatusBarString = buffer;
    SetMessageText((LPCTSTR)m_sStatusBarString);
    return;
}

```

在应用程序中使用这些函数的示范代码如下:

```

CString sReplacementText = "替换默认的文本!";
((CMainFrame*) AfxGetMainWnd()) -> StatusBarMessage(sReplacementText. Get-
Buffer(0));

```

9.6 在状态栏中使用控件

工具栏中能够使用控件,那么在状态栏中也同样能使用。本节将设计一个通用控件状态栏类,用于管理各种插入状态栏中的控件。

插入状态栏中的每个控件都被认为是它的子窗口,并且其 ID 与窗格 ID 相同。这意味着,状态栏中不可能同时创建两个相同的窗口。

9.6.1 设计通用控件状态栏类

当向状态栏中添加控件时,需要首先创建控件窗口,然后设置它们的位置和外形。而

这些操作对于每个控件来说,都是必需的,并且都具有相似处理流程。因此,设计一个类将这些流程封装起来,不仅能够减少重复劳动,还有利于提高功能的扩展性。

清单 9-13 所示为 CStatusControl 类的定义:

清单 9-13 CStatusControl 类定义

```
class CStatusControl : public CWnd
{
public:
    friend class CStatusEdit;
    friend class CStatusProgress;
    friend class CStatusStatic;
    friend class CStatusCombo;
    CStatusControl();
    BOOL Create(LPCTSTR classname, CStatusBar * parent, UINT id, DWORD style);
    void Reposition();
    virtual ~CStatusControl();
protected:
    static void reposition(CWnd * wnd);
    static BOOL setup(CStatusBar * parent, UINT id, CRect & r);
    DECLARE_MESSAGE_MAP()
}
```

类还定义了 4 个友元类: CStatusEdit、CStatusProgress、CStatusStatic 和 CStatusCombo。它们分别用于管理状态栏中的编辑控件、进度控件、静态控件和组合框控件。之所以将它们定义为 CStatusControl 的友元类,使为了使这些类能够使用其中定义的方法。读者如果感兴趣,可以为它添加更多的友元类,以扩展其功能。

setup 函数为静态函数,它将被其友元类用于计算目标矩形。该函数将判断指定窗格是否被其他窗口裁剪;如果被裁剪,则 GetItemRect 函数将返回零矩形(0,0,0,0)。此时,Setup 函数将紧挨着状态栏的右上角创建一个很窄的矩形。然后在 WM_SIZE 消息的相应函数中重新设置窗口尺寸。如果创建的窗口是不可见的,则函数将返回 FALSE。清单 9-14 所示为 setup 函数的源代码:

清单 9-14 setup() 函数

```
BOOL CStatusControl::setup(CStatusBar * parent, UINT id, CRect & r)
{
    int i = parent -> CommandToIndex(id);

    parent -> GetItemRect(i, &r);
    parent -> SetPaneText(i, "");

    if(r.IsRectEmpty())
    { /* offscreen */
        CRect r1;
        parent -> GetWindowRect(&r1); // 得到父窗口(状态栏)的宽度
        r.left = r1.right + 1;
        r.top = r1.top;
        r.right = r1.right + 2;
    }
}
```



```

        r.bottom = r1.bottom;
        return FALSE;
    }
    return TRUE;
}

```

reposition 函数为静态函数,它将被友元类的 Reposition 函数调用,以进行实际的窗口定位。由于 Windows 将窗格 ID 作为控件 ID,因此我们很容易定位窗格的位置和尺寸。在本函数中并不需要像 setup 函数那样,处理零矩形情况,因为此时窗口已经存在(可能是 setup 函数中创建的窄矩形)。清单 9-15 所示为 reposition 函数的源代码:

清单 9-15 reposition() 函数

```

void CStatusControl::reposition(CWnd * wnd)
{
    if(wnd == NULL || wnd->m_hWnd == NULL)
        return;
    UINT id = ::GetWindowLong(wnd->m_hWnd, GWL_ID);
    CRect r;

    CStatusBar * parent = (CStatusBar *)wnd->GetParent();
    int i = parent->CommandToIndex(id);
    parent->GetItemRect(i, &r);
    wnd->SetWindowPos(&wndTop, r.left, r.top, r.Width(), r.Height(), 0);
}

```

Create 函数能够使用控件类名和状态标志,来创建由任意窗口类管理的窗口。清单 9-16 所示为 Create 函数的源代码:

清单 9-16 Create() 函数

```

BOOL CStatusControl::Create(LPCTSTR classname, CStatusBar * parent, UINT id,
DWORD style)
{
    CRect r;
    setup(parent, id, r);
    return CWnd::Create(classname, NULL, style | WS_CHILD, r, parent, id);
}

```

Reposition 函数负责为 CStatusControl 类提供输出接口,而其实际功能由 reposition 完成。清单 9-17 所示为 Reposition 函数的源代码:

清单 9-17 Reposition() 函数

```

void CStatusControl::Reposition()
{
    reposition(this);
}

```

9.6.2 设计控件友元类

本节中将向读者介绍 CStatusControl 类的 4 个友元类的设计。它们的定位、窗口创建

等工作已经在 CStatusControl 中实现了,本身只需完成具体控件的创建即可。

1. CStatusStatic 类

该类负责管理状态栏中的静态控件,其定义如清单 9-18 所示:

清单 9-18 CStatusStatic 类定义

```
class CStatusStatic : public CStatic
{
public:
    CStatusStatic();
    BOOL Create(CStatusBar * parent, UINT id, DWORD style);
public:
    __inline void Reposition() { CStatusControl::reposition(this); }
public:
    virtual ~CStatusStatic();
};
```

其中内联函数 Reposition 通过 CStatusControl::reposition 完成功能。而 Create 函数则负责创建静态控件本身,其源代码如清单 9-19 所示:

清单 9-19 Create()函数

```
BOOL CStatusStatic::Create(CStatusBar * parent, UINT id, DWORD style)
{
    CRect r;
    CStatusControl::setup(parent, id, r);
    BOOL result = CStatic::Create(NULL, style | WS_CHILD, r, parent, id);
    if(! result)
        return FALSE;
    CFont * f = parent->GetFont();
    SetFont(f);
    return TRUE;
}
```

2. CStatusEdit 类

该类负责管理状态栏中的编辑控件,其定义如清单 9-20 所示:

清单 9-20 CStatusStatic 类定义

```
class CStatusEdit : public CEdit
{
public:
    CStatusEdit();
    BOOL Create(CStatusBar * parent, UINT id, DWORD style);
public:
    __inline void Reposition() { CStatusControl::reposition(this); }
public:
    virtual ~CStatusEdit ();
};
```

其中内联函数 `Reposition` 通过 `CStatusControl::reposition` 完成其功能。`Create` 函数则负责创建编辑控件本身,其源代码如清单 9-21 所示:

清单 9-21 `Create()` 函数

```

BOOL CStatusEdit::Create(CStatusBar * parent, UINT id, DWORD style)
{
    CRect r;
    CStatusControl::setup(parent, id, r);
    BOOL result = CEdit::Create(style | WS_CHILD, r, parent, id);
    if(! result)
        return FALSE;
    CFont * f = parent -> GetFont();
    SetFont(f);
    return TRUE;
}

```

3. `CStatusProgress` 类

该类负责管理状态栏中的进度控件,其定义如清单 9-22 所示:

清单 9-22 `CStatusProgress` 类定义

```

class CStatusProgress : public CProgressCtrl
{
public:
    CStatusProgress();
    BOOL Create(CStatusBar * parent, UINT id, DWORD style);
public:
    __inline void Reposition() { CStatusControl::reposition(this); }
public:
    virtual ~CStatusProgress();
};

```

其中内联函数 `Reposition` 通过 `CStatusControl::reposition` 完成功能。而 `Create` 函数则负责创建进度控件本身,其源代码如清单 9-23 所示:

清单 9-23 `Create()` 函数

```

BOOL CStatusEdit::Create(CStatusBar * parent, UINT id, DWORD style)
{
    CRect r;
    CStatusControl::setup(parent, id, r);
    BOOL result = CEdit::Create(style | WS_CHILD, r, parent, id);
    if(! result)
        return FALSE;
    CFont * f = parent -> GetFont();
    SetFont(f);
    return TRUE;
}

```

4. CStatusCombo 类

该类负责管理状态栏中的组合框控件,其定义如清单 9-24 所示:

清单 9-24 CStatusCombo 类定义

```
class CStatusCombo : public CComboBox
{
public:
    CStatusCombo();
    BOOL Create(CStatusBar * parent, UINT id, DWORD style);
public:
    __inline void Reposition() { CStatusControl::reposition(this); }
public:
    virtual ~CStatusCombo();
    int maxlen;
protected:
    //||AFX_MSG(CStatusCombo)
    afx_msg void OnDropdown();
    //||AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

其中内联函数 Reposition 通过 CStatusControl::reposition 完成功能;maxlen 成员则指定了组合框中能够插入的最大选项数。由于组合框还涉及到下拉选择的一些问题,因此添加了对下拉消息的处理函数 OnDropdown。Create 函数负责创建组合框控件本身,其源代码如清单 9-25 所示:

清单 9-25 Create() 函数

```
BOOL CStatusCombo::Create(CStatusBar * parent, UINT id, DWORD style)
{
    CRect r;
    CStatusControl::setup(parent, id, r);
    BOOL result = CComboBox::Create(style | WS_CHILD, r, parent, id);
    if(! result)
        return FALSE;
    CFont * f = parent->GetFont();
    SetFont(f);
    return TRUE;
}
```

OnDropdown 负责处理下拉选择(用户按下组合框右侧的箭头)消息,并计算下拉列表框将出现的位置。其源代码如清单 9-26 所示:

清单 9-26 OnDropdown() 函数

```
void CStatusCombo::OnDropdown()
{
    int n = GetCount();
    n = max(n, 2);
```

```

int ht = GetItemHeight(0);
CRect r;
GetWindowRect(&r);
if (maxlen > 0)
    n = max(maxlen, 2);

CSize sz;
sz.cx = r.Width();
sz.cy = ht * (n + 2);

if (maxlen == 0)
{
    /* screen limit */
    if (r.top - sz.cy < 0 || r.bottom + sz.cy > ::GetSystemMetrics(SM_CYSCREEN))
    {
        /* invoke limit */
        // 计算下拉列表框能够出现的最大尺寸
        int k = max((r.top / ht), (::GetSystemMetrics(SM_CYSCREEN) - r.bottom) / ht);

        // 计算可用空间
        int ht2 = ht * k;
        sz.cy = min(ht2, sz.cy);
    }
}

SetWindowPos(NULL, 0, 0, sz.cx, sz.cy, SWP_NOMOVE | SWP_NOZORDER);
}

```

9.6.3 应用实例

本节将使用前两节设计的状态栏控件类,设计包含不同控件的状态栏,如图 9-2 所示。



图 9-2 包含不同控件的状态栏

首先创建单文档/视图结构的应用程序,其中每一步都接受默认选项。下面以在状态栏中添加进度控件为例,说明这些类的使用方法:

(1) 在资源编辑器中添加一个字符串,并将其 ID 设置为 ID_INDICATOR_PROGRESS。将字符串内容设置为一些空格,它们将被用来确定状态栏窗格的初始宽度。

(2) 在 CMainFrame 类的实现代码中,找到窗格数组 indicators,并将 ID_INDICATOR_PROGRESS 添加进去。注意该 ID 应该被添加在 ID_SEPARATOR 之后。

(3) 将 StatusControl.h 和 StatusProgress.h 文件包含到 CMainFrame 类中。

(4) 在 CMainFrame 类中定义 CStatusProgress 类型的变量 progressbar,即进度控件对象。

(5) CMainFrame 类的 OnCreate 函数调用 progressbar 对象的 Create 函数(注意必须在创建状态栏的代码后)来创建进度控件。当然,用户也可在必要的时候创建或销毁,这与应用程序的具体情况有关。创建进度控件的示范代码如下:


```
progressbar.Create(&m_wndStatusBar, ID_INDICATOR_PROGRESS, WS_VISIBLE |
    PBS_SMOOTH);
```

注意 WS_CHILD 风格在 CStatusControl 类中已经被自动添加了。当希望销毁控件时, 只要调用如下代码即可:

```
progressbar.DestroyWindow();
```

(6) 调用进度控件的成员函数, 对其进度操作, 例如:

```
progressbar.SetRange(0, 500);
progressbar.StepIt();
progressbar.SetPos(value);
```

(7) 在 CMainFrame 类中处理 OnSize 消息处理函数, 该函数对所有状态栏控件进行重新定位。清单 9-27 所示为 OnSize 函数的源代码:

清单 9-27 OnSize() 函数

```
void CMainFrame::OnSize(UINT nType, int cx, int cy)
{
    CMDIFrameWnd::OnSize(nType, cx, cy);
    progressbar.Reposition();
}
```

这样就完成了向状态栏中添加进度控件的操作。添加其他控件的操作也类似, 请读者参考配套光盘中 chap9/statuscontrol 下的源代码。

如果进度控件是用于显示背景线程的进度的, 那么最好再使背景线程使用自定义消息来通告主窗口更新进度控件, 因为从其他线程直接操作控件会导致应用程序的锁定。例如当 GUI 线程处于阻塞状态时, 其他线程的 SendMessage 调用将等待直到 GUI 线程重新运行, 而这将导致死锁状态。

9.6.4 使用自定义消息响应状态栏控件动作

使用自定义消息的一般步骤如下:

(1) 选择自定义消息, 并将其定义在一个 include 文件中, 例如:

```
#define UWM_STEP_PROGRESS (WM_APP + 103)
#define UWM_SET_PROGRESS (WM_APP + 104)
```

(2) 在消息映射中添加这些消息的入口:

```
ON_MESSAGE(UWM_STEP_PROGRESS, OnStepProgress)
ON_MESSAGE(UWM_SET_PROGRESS, OnSetProgress)
```

(3) 在 MainFrm.h 中, 以 afx_msg 形式添加下列消息处理函数:

```
afx_msg LRESULT OnStepProgress(WPARAM, LPARAM);
afx_msg LRESULT OnSetProgress(WPARAM, LPARAM);
```

(4) 在 MainFrm.cpp 中添加这些函数的实现代码,如清单 9-28 和 9-29 所示:

清单 9-28 OnStepProgress() 函数

```
LRESULT CMainFrame::OnStepProgress(WPARAM, LPARAM)
{
    progressbar.StepIt();
    return 0;
}
```

清单 9-29 OnSetProgress() 函数

```
LRESULT CMainFrame::OnSetProgress(WPARAM wParam, LPARAM)
{
    progressbar.SetPos((int)wParam);
    return 0;
}
```

(5) 从其他线程中激活控件操作,这一般是通告 PostMessage(而不是 SendMessage 函数)实现:

```
AfxGetMainWnd()->PostMessage(UWM_STEP_PROGRESS);
```

9.6.5 使用注册窗口消息响应状态栏控件动作

使用自定义消息(以 WM_USER 为基的消息)的问题之一就是不能保证消息的唯一性。这样如果您使用别人的有相同自定义消息的 DLL 时,就会出现麻烦。当然您的应用程序依然能够在 Windows 下使用,因为每个控件使用其自己定义的消息。现在鼓励程序员使用 WM_APP 来取代 WM_USER,但是这只解决了与 Microsoft 公司产品代码间的冲突,而不能解决与其他程序员编写的 DLL 之间的冲突。实际上有另外一个解决之道,就是使用注册窗口消息,并且这种方法并不比自定义消息更难。

(1) 选择消息名称,并将其包括在一个定义文件中:

```
#define UWM_STEP_PROGRESS_MESSAGE T("UWM_STEP_PROGRESS")
#define UWM_SET_PROGRESS_MESSAGE T("UWM_SET_PROGRESS")
```

如果觉得还有可能发生冲突,可以选择以下的定义方式:

```
#define UWM_STEP_PROGRESS_MESSAGE
    _T("UWM_STEP_PROGRESS-152C2190-A98C-11d2-838D-886273000000")
```

(2) 在每个线程的实现代码(包括 MainFrm.cpp)中包含以上定义文件。

(3) 在每个线程的实现代码(包括 MainFrm.cpp)中使用如下方式注册消息:

```
const WORD UWM_STEP_PROGRESS =
    ::RegisterWindowMessage(UWM_STEP_PROGRESS_MESSAGE);
const WORD UWM_SET_PROGRESS =
    ::RegisterWindowMessage(UWM_SET_PROGRESS_MESSAGE);
```

(4) 在 CMainFrame 类的消息映射中,添加如下消息入口:

```
ON_REGISTERED_MESSAGE(UWM_STEP_PROGRESS, OnStepProgress)  
ON_REGISTERED_MESSAGE(UWM_SET_PROGRESS, OnSetProgress)
```

至于控件消息的处理,则与自定义消息一致。

本章小结

本章主要介绍的是状态栏编程的基本思路和方法。通过本章的学习,读者应该达到以下几点:

- 掌握 CStatusBar 类的使用。
- 掌握定制状态栏的方法和思路。

第 10 章 框架窗口

本章将向读者介绍定制框架窗口的一些技巧。实际上,使用前面几章介绍的方法就能改变窗口组成控件,从而使应用程序看起来更加具有吸引力。而且每个控件都是窗口的一种,定制控件形状、颜色、字体等方法 and 原则同样也适用于框架窗口。因此这里将不对这些问题再加讨论,而是介绍其他包装方式,包括:增加闪屏效果、添加窗口背景等。

本章要点:

- 改变窗口效果;
- 增加闪屏效果;
- 添加窗口背景。

10.1 改变窗口效果

AppWizard 会为每个应用程序生成默认的图标、标题等窗口效果,但是这些效果显然不能完全令人满意。为了使自己的应用程序看起来更加美观,我们经常需要改变这些窗口效果。

10.1.1 应用程序的默认图标

使用 Visual C++ 创建应用程序时,AppWizard 会为用户生成一个默认的 MFC 图标。但是这个图标既不美观,也缺乏个性,因此常常需要修改应用程序的默认图标。

在 ResourceView 中,单击 Icon 项左边的“+”号,然后将 AppWizard 为程序生成的 IDR_MAINFRAME 和 IDR_PAINTETYPE 删除,最后再插入用户所选择的新图标。不过需要注意的是,一定要将新插入的图标 ID 分别设置为 IDR_MAINFRAME 和 IDR_PAINTETYPE 类型。其中 IDR_MAINFRAME 是用于框架窗口的图标,而 IDR_PAINTETYPE 则是用于子窗口的图标。图 10-1 为改变图标后的应用程序运行后的标题栏。



图 10-1 改变图标后的应用程序标题栏

10.1.2 修改窗口的默认风格

重载 `CWnd::PreCreateWindow` 函数。

通过改变风格参数可以定制不同样式的窗口,有时能获得意想不到的效果,一般通过重载 `CWnd::PreCreateWindow` 函数,并修改 `CREATESTRUCT` 结构来指定窗口风格和其他创建信息,该函数的原型为:

```
virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
```

其中窗口风格结构 `CREATESTRUCT` 的定义如下:

```
typedef struct tagCREATESTRUCT
{
    LPVOID lpCreateParams; //窗口创建参数
    HANDLE hInstance;      //窗口句柄
    HMENU hMenu;           //窗口使用的菜单
    HWND hwndParent;       //窗口的父窗口
    Int cy;                //窗口的高度
    Int cx;                //窗口的宽度
    Int y;                 //窗口的左上角 x 坐标
    Int x;                 //窗口的左上角 y 坐标
    LONG style;            //窗口风格
    LPCSTR lpszName;       //窗口名称
    LPCSTR lpszClass;      //管理窗口的类名
    DWORD dwExstyle;       //窗口的扩展风格
} CREATESTRUCT;
```

通过修改 `cs` 中相应变量的值就能够改变框架窗口的风格,窗口的风格参数 `style` 可以为下面选项的组合:

- `WS_BORDER`: 选择该选项将创建具有边框的窗口。
- `WS_CAPTION`: 选择该选项将创建具有标题栏的窗口,但该选项只对 `WS_BORDER` 风格有效。
- `WS_CHILD`: 选择该选项将创建一个子窗口,但该选项对 `WS_POPUP` 风格无效。
- `WS_CLIPCHILDREN`: 选择该选项用于创建父窗口。
- `WS_CLIPSIBLINGS`: 选择该选项使当子窗口中接受到重绘消息时,不会使其他被覆盖的子窗口也进行重绘。该选项只对 `WS_CHILD` 风格有效。
- `WS_DISABLED`: 选择该选项将创建一个初始处于被禁止状态的窗口。
- `WS_DLGFRAAME`: 选择该选项将创建没有标题栏的具有双层边框的窗口。
- `WS_GROUP`: 选择该选项将能够使用箭头按钮在指定控件组中进行选择,指定控件组时只需将控件组的第一个控件指定为 `WS_GROUP` 风格即可,控件组中的其他控件不能选择该选项,否则将创建第二个控件组。
- `WS_HSCROLL`: 选择该选项将创建具有水平滚动条的窗口。
- `WS_MAXIMIZE`: 选择该选项将使窗口在创建时最大化。

- **WS_MAXIMIZEBOX**: 选择该选项将创建具有最大化按钮的窗口。
- **WS_MINIMIZE**: 选择该选项将使窗口在创建时最小化, 该选项只对 **WS_OVERLAPPED** 风格有效。
- **WS_MINIMIZEBOX**: 选择该选项将创建具有最小化按钮的窗口。
- **WS_OVERLAPPED**: 选择该选项将创建重叠的窗口, 一般具有该风格的窗口有边框和标题。
- **WS_OVERLAPPEDWINDOW**: 选择该选项将创建具有 **WS_OVERLAPPED**、**WS_CAPTION**、**WS_SYSMENU**、**WS_THICKFRAME**、**WS_MINIMIZEBOX** 和 **WS_MAXIMIZEBOX** 风格的重叠窗口。
- **WS_POPUP**: 选择该选项将创建弹出式窗口。
- **WS_POPUPWINDOW**: 选择该选项将创建具有 **WS_BORDER**、**WS_POPUP** 和 **WS_SYSMENU** 风格的弹出窗口。该风格必须与 **WS_CAPTION** 风格一起使用, 以使控制菜单能够显示。
- **WS_SYSMENU**: 选择该选项将创建在标题栏中带有控制菜单按钮的窗口。
- **WS_TABSTOP**: 选择该选项将使用户能够通过 Tab 键在具有该风格的控件间切换。
- **WS_THICKFRAME**: 选择该选项将创建具有能够用来调整窗口尺寸的粗框架。
- **WS_VISIBLE**: 选择该选项将创建初始处于可见状态的窗口。
- **WS_VSCROLL**: 选择该选项将创建具有垂直滚动条的窗口。

窗口的扩展风格参数 **dwExstyle** 可以为下面选项的组合:

- **WS_EX_ACCEPTFILES**: 选择该选项将创建允许拖放文件的窗口。
- **WS_EX_CLIENTEDGE**: 选择该选项将创建具有 3D 外观的窗口, 即具有凹陷风格的边框。
- **WS_EX_CONTEXTHELP**: 选择该选项将使窗口的标题栏中包括一个问号标记。当用户单击该问号标记时, 光标将变为问号箭头形式, 此时再用光标单击子窗口就会得到帮助信息。
- **WS_EX_CONTROLPARENT**: 选择该选项允许用户使用 Tab 键在子窗口之间切换。
- **WS_EX_DLGMODALFRAME**: 选择该选项将创建具有双层边框的窗口, 如果在 **style** 风格选项中指定了 **WS_CAPTION** 风格则窗口还可具有标题栏。
- **WS_EX_LEFT**: 选择该选项将使子窗口具有左对齐风格。
- **WS_EX_LEFTSCROLLBAR**: 选择该选项将使垂直滚动条位于客户区的左侧。
- **WS_EX_LTRREADING**: 选择该选项将使窗口具有由左到右的窗口文本阅读风格。
- **WS_EX_MDICHILD**: 选择该选项将创建 MDI 子窗口。
- **WS_EX_NOPARENTNOTIFY**: 选择该选项将使所创建的子窗口在生成或销毁时不向其父窗口发送 **WM_PARENTNOTIFY** 消息。
- **WS_EX_OVERLAPPEDWINDOW**: 选择该选项将创建具有 **WS_EX_CLIENTEDGE** 和 **WS_EX_WINDOWEDGE** 组合风格的窗口。

- **WS_EX_PALETTEWINDOW**: 选择该选项将创建具有 **WS_EX_WINDOWEDGE** 和 **WS_EX_TOPMOST** 组合风格的窗口。
- **WS_EX_RIGHT**: 选择该选项将使窗口具有右对齐风格。
- **WS_EX_RIGHTSCROLLBAR**: 选择该选项将使垂直滚动条位于客户区的右侧。
- **WS_EXRTLREADING**: 选择该选项将使窗口文本的阅读顺序为由右到左。
- **WS_EX_STATICEDGE**: 选择该选项将创建具有三维边框风格的窗口,通常该风格用于创建不接受用户输入的项。
- **WS_EX_TOOLWINDOW**: 选择该选项将创建工具窗口,工具窗口能够作为不固定的工具栏。
- **WS_EX_TOPMOST**: 选择该选项将使所创建的窗口位于所有非顶端窗口之上,即使窗口处于不激活状态也是如此。应用程序可以通过 **SetWindowPos** 成员函数来改变该属性。
- **WS_EX_TRANSPARENT**: 选择该选项将创建具有透明风格的窗口,即任何在该窗口之后的窗口都不会被遮住。具有这种风格的窗口只有在所有在其之后的姊妹窗口都被更新后,才会接收到 **WM_PAINT** 消息。
- **WS_EX_WINDOWEDGE**: 选择该选项将创建具有凸起风格边框的窗口。

清单 10-1 所示的代码使被创建的窗口拥有系统菜单以及最大、最小化按钮等。

清单 10-1 PreCreateWindows() 函数

```

BOOL CMainFrame::PreCreateWindow(CREATESTRUCT &cs)
{
    cs.x = cs.y = 0;
    cs.cx = GetSystemMetrics(SM_CXSCREEN/2);
    cs.cy = GetSystemMetrics(SM_CYSCREEN/2);
    cs.style = WS_OVERLAPPED | WS_CAPTION | FWS_ADDTOTITLE
               | WS_THICKFRAME | WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX |
               WS_MAXIMIZE;
    return CMDIFramewnd::PreCreateWindow(cs);
}

```

10.1.3 改变窗口标题

调用 **CWnd::SetWindowText** 函数可以改变任何窗口(包括控件)的标题,示范代码如下:

```

// 设置主框架窗口的标题
AfxGetMainWnd() -> SetWindowText(_T("我的标题"));
//设置视图窗口窗口标题
GetParentFrame() -> SetWindowText(_T("我的子窗口标题"));

```

如果需要经常修改窗口的标题(注意控件也是窗口),应该考虑使用半文档化的函数 **AfxSetWindowText**。该函数在 **AFXPRIV.H** 中说明,在 **WINUTIL.CPP** 中实现,但是在联机帮

助中找不到它,它在 AFXPRIV.H 中半文档化,在以后发行的 MFC 中将文档化。AfxSetWindowText 函数的实现代码如清单 10-2 所示。

清单 10-2 AfxSetWindowText()函数

```
void AFXAPI AfxSetWindowText (HWND hWndCtrl , LPCTSTR lpszNew )
{
    int nNewLen = lstrlen (lpszNew);
    TCHAR szOld [256];
    if (nNewLen > _countof (szOld) || ::GetWindowText (hWndCtrl, szOld , _countof
        (szOld) != nNewLen
        || lstrcmp (szOld , lpszNew) != 0)
    {
        ::SetWindowText(hWndCtrl , lpszNew )
    }
}
```

10.1.4 改变窗口位置和排列

使用软件时,我们有时会发现一些总处于别的窗口之上的窗口,即使它失去了焦点也是如此。要实现这一点并不难,只要调用 SetWindowPos 函数,并指定窗口具有最顶风格即可,示范代码如清单 10-3 所示:

清单 10-3 TopMost()函数

```
void TopMost(CWnd * pWnd)
{
    ASSERT_VALID(pWnd);
    pWnd -> SetWindowPos(pWnd -> GetStyle() & WS_EX_TOPMOST)? &wndNoTopMOST:
        &wndTopMost,0,0,0,0,SSP_NOSIZE|WSP_NOMOVE);
}
```

调用 SetWindows 函数,还能够在程序运行时通过代码改变窗口的的位置。只要在调用该函数时,指定 SWP_NOSIZE 标志即可。移动的目的位置是相对于父窗口的(顶层窗口与屏幕有关)。例如下面的示范代码将窗口移动到其父窗口中的(100,100)位置:

```
SetWindowPos (NULL, 100 , 100 , 0 , 0 , SWP_NOSIZE |SWP_NOAORDER)
```

调用 CWnd::SetWindowPos 函数并指定 SWP_NOMOVE 标志,能够在运行时改变窗口的大小。也可调用 CWnd::MoveWindow 函数但必须指定窗口的位置。下面的示范代码将窗口的尺寸扩大了两倍:

```
// 得到窗口的当前尺寸
CRect reWindow;
GetWindowRect (reWindow );
SetWindowPos (NULL , 0 , 0 , reWindow . Width ( ) * 2,reWindow . Height ( ) * 2, SWP_
    NOMOVE |SWP_NOZORDER );
```

10.1.5 改变窗口形状

有关改变窗口形状的话题,第2章已经有过介绍。虽然当时的处理对象是按钮,但当时所用的原则和方法同样适用于框架窗口。为了使读者更加清楚,这里依然给出一些说明。

可以通过 SetWindowRgn 函数为窗口设置区域,从而改变窗口的形状。该函数将绘画和鼠标消息限定在窗口的一个指定的区域,实际上使窗口成为指定的不规则形状。

下面以创建不规则形状的对话框为例,介绍设置窗口区域的方法。使用 AppWizard 创建一个基于对话框的应用程序,并使用资源编辑器从主对话资源中删除所有默认控件、标题以及边界。

在对话框类增加一个 CRgn 数据成员,以后要使用该数据成员创建窗口区域:

```
private :
    CRgn m_rgn;          // 窗口区域
```

修改 OnInitDialog 函数创建一个椭圆区域,并调用 SetWindowRgn 函数将该区域分配给窗口。清单 10-4 所示为 OnInitDialog 函数的源代码:

清单 10-4 OnInitDialog()函数

```
BOOL CRoundDlg::OnInitDialog ( )
{
    CDialog::OnInitDialog ( );

    // 得到对话框的尺寸
    CRect rcDialog;
    GetClientRect (rcDialog );

    // 创建区域,并将其赋予窗口
    m_rgn . CreateEllipticRgn (0 , 0 , rcDialog.Width ( ) , rcDialog.Height ( ) );
    SetWindowRgn (GetSafeHwnd ( ) , (HRGN) m_rgn , TRUE );
    return TRUE
}
```

通过创建区域和调用 SetWindowRgn 函数,已经创建一个不规则形状的窗口,清单 10-5 所示的 OnPaint 函数中的代码能够使窗口形状看起来像一个球形:

清单 10-5 OnPaint()函数

```
void CRoundDlg::OnPaint ( )
{
    CPaintDC dc (this);

    // 绘制椭圆,但由于使用的是空画笔,因而该椭圆不会具有任何边界
    dc . SelectStockObject (NULL_PEN);
    // 得到球体颜色的 RGB 组份
    COLORREF color = RGB (0 , 0 , 255);
    BYTE byRed = GetRValue (color);
```

```

BYTE byGreen = GetGValue (color);
BYTE byBlue = GetBValue (color);

// 得到窗口的尺寸
CRect rect;
GetClientRect (rect);

// 得到最少的单位数
int nUnits = min (rect.right , rect.bottom );

// 计算水平和垂直步长
float fltStepHorz = (float) rect.right /nUnits;
float fltStepVert = (float) rect.bottom /nUnits;

int nEllipse = nUnits/3; // 计算绘制量
int nIndex;

CBrush brush;
// 用于填充椭圆的颜色
CBrush * pBrushOld ;
for (nIndex = 0 ; nIndex < nUnits/nEllipse ; nIndex++)
{
brush . CreateSolidBrush (RGB ( ( nIndex * byRed ) /nEllipse ).( nIndex * byGreen )
/nEllipse ),
( ( nIndex * byBlue)/nEllipse ) );

pBrushOld= dc .SelectObject (&brush);
// 绘制椭圆
dc .Ellipse ( (int) fltStepHorz * 2, (int) fltStepVert * nIndex , rect. right -
( (int) fltStepHorz * nIndex )+ 1,
rect . bottom-( (int) fltStepVert * (nIndex * 2) ) +1);

brush.DeselectObject ( );
}
}

```

最后,处理 WM_NCHITTEST 消息,使单击窗口的任何位置时能移动窗口。清单 10-6 所示即为 OnNchitTest 函数的源代码:

清单 10-6 OnNchitTest()函数

```

UINT CRoundDlg::OnNchitTest (Cpoint point )
{
UINT nHitTest = CDialog::OnNcHitTest (point);
return (nHitTest == HTCLIENT)? HTCAPTION: nHitTest;
}

```

10.2 添加闪屏效果

所谓闪屏就是在应用程序启动时,出现的类似于封面的图形或对话框。添加闪屏效果非常简单,只要选择 Project|Add To Project|Components and Controls 菜单命令,这时弹出

Components and Controls Gallery 对话框。双击其中的 Developer Studio Components 文件夹,选择 Splash Screen 组件,单击 Insert 按钮即可。这时,Visual C++ 会为应用程序添加 CSplashWnd 类,该类专门用于管理闪屏。类中函数及其功能如表 10-1 所示。

表 10-1 CSplashWnd 类中的函数及其功能

函数	功能
Create	响应 Create 消息,创建并注册闪屏窗口
~ CsplashWnd	闪屏类的析构函数
CsplashWnd	闪屏类的构造函数
EnableSplashScreen	允许闪屏窗口显示
HideSplashScreen	隐藏闪屏窗口,并销毁窗口对象
OnCreate	响应 WM_CREATE 消息,设置窗口显示
OnTimer	调用 HideSplashScreen 函数,以隐藏闪屏窗口
OnPaint	绘制窗口
PostNcDestroy	销毁闪屏类对象
PreTranslateAppMessage	获取键盘和鼠标消息并以隐藏闪屏窗口
ShowSplashScreen	显示闪屏窗口

这时还需向应用程序中插入 ID 为 IDB_SPLASH 的位图,以作为闪屏窗口的显示内容。用户可选择和编辑自己喜欢的图像。

如果读者希望改变闪屏持续的时间,可以编辑闪屏类的 OnCreate 函数,修改其中的定时器设置。清单 10-7 所示代码将闪屏效果持续时间设置为 2 秒:

清单 10-7 OnCreate()函数

```
int CSplashWnd::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CWnd::OnCreate(lpCreateStruct) == -1)
        return -1;

    // 使闪屏出现在屏幕中央
    CenterWindow();

    // 设置闪屏持续的时间
    SetTimer(1, 2000, NULL);

    return 0;
}
```

首先在 CPainterApp 类的头文件中包含闪屏类的头文件 Splash.h,然后再向其 InitInstance 中添加下面的语句,就能够显示闪屏效果:

```
CSplashWnd::ShowSplashScreen(pMainFrame);
```

10.3 添加窗口背景

关闭主框架中的所有活动视图后会出现灰色的窗口背景。此时如果能够在背景上绘

制一些图案,那么就会大大改善窗口的外观。

对于多文档/视图结构的应用程序来说,其框架窗口由 MFC 类库中的 `CMDIFrameWnd` 类管理,在该类的源代码中声明了如下变量:

```
public:
    HWND m_hWndMDIClient;    // MDI 客户窗口句柄
```

该变量即为管理框架窗口客户区的句柄。默认情况下,框架窗口的背景为灰色。改变这个背景,实际就是在框架窗口的客户区背景中进行绘制。为了便于管理,一般重新创建一个由 `CWnd` 类派生的窗口管理类,利用该类实现窗口背景重绘。然后再使用 `SubclassWindow` 函数,将该窗口管理类与 `m_hWndMDIClient` 句柄相联系。

为应用程序添加窗口背景的步骤如下:

- (1) 使用 ClassWizard 向应用程序中添加 `CWnd` 派生窗口类,并将类名定为 `CMdiClient`。
- (2) 每当接收到 `WM_ERASEBKGND` 消息时,应用程序就会重绘窗口背景。因此在类中只要加入响应 `WM_ERASEBKGND` 消息的处理函数即可。清单 10-8 为消息函数 `OnEraseBkgnd` 的源代码:

清单 10-8 `OnEraseBkgnd()` 函数

```
BOOL CMdiClient::OnEraseBkgnd(CDC * pDC)
{
    // TODO: Add your message handler code here and/or call default
    CWnd::OnEraseBkgnd(pDC);

    CRect rect;
    GetClientRect(&rect);
    CDC dc;
    dc.CreateCompatibleDC(pDC);
    dc.SelectObject(background);

    BITMAP bm;
    background.GetBitmap(&bm);

    pDC->StretchBlt(0,0,rect.right-rect.left,rect.bottom-rect.top,
        &dc,0,0,463,451,SRCCOPY);
    CString text1("背景图像");
    CString text2("这是背景图像");
    CString text3("背景图像可以随便选择");
    int x = rect.right-dc.GetTextExtent(text2).cx;
    int y = rect.bottom-8-20;
    pDC->SetBkMode(TRANSPARENT);
    pDC->SetTextColor(RGB(255,255,255));
    pDC->TextOut(x,y,text3);
    pDC->TextOut(x,y-20,text2);
    pDC->TextOut(x,y-40,text1);
    pDC->SetTextColor(RGB(0,0,0));
    pDC->TextOut(x-1,y-1,text3);
    pDC->TextOut(x-1,y-21,text2);
    pDC->TextOut(x-1,y-41,text1);
}
```

```
return TRUE;
```

该函数将 background 所代表的位图(IDB_BACKGROUND)选入临时设备环境中,然后使用 StretchBlt 将该位图拷贝并充满背景。然后在窗口的右下角输出了 3 行文本。输出前,先调用设备环境的 GetTextExtent 函数来确定文本长度,并据此设置合适的输出位置。

(3) 框架窗口管理类 CMainFrame 中的 OnCreate 函数调用 SubclassWindow 函数,将该窗口管理类与 m_hWndMDIClient 句柄相联系。实现代码如下:

```
m_pmdiclient->SubclassWindow(m_hWndMDIClient);
```

读者现在如果改变窗口的大小,就会发现背景图案不能随着窗口尺寸的改变而改变。例如,当减小窗口尺寸时,只能在窗口背景中显示部分背景图案。由于当窗口的尺寸改变时,应用程序就会接收到 WM_SIZE 消息。因此为了解决这个问题,可以向 CMdiClient 类中添加对 WM_SIZE 消息的处理函数,并在该函数中完成窗口背景的重绘。OnSize 消息处理函数的源代码如清单 10-9 所示:

清单 10-9 OnSize()函数

```
void CMdiClient::OnSize(UINT nType, int cx, int cy)
{
    CWnd::OnSize(nType, cx, cy);

    // TODO: Add your message handler code here
    RedrawWindow(NULL, NULL,
        RDW_INVALIDATE | RDW_ERASE | RDW_ERASENOW | RDW_ALLCHILDREN);
}
```

其中 RedrawWindow 函数可以重绘窗口的客户区,其原型为:

```
BOOL RedrawWindow( LPCRECT lpRectUpdate = NULL, CRgn* prgnUpdate = NULL, UINT
    flags = RDW_INVALIDATE | RDW_UPDATENOW | RDW_ERASE );
```

由于指定了 RDW_ERASE 标志,因此调用 RedrawWindow 函数就会发送 WM_ERASE 消息,从而触发 OnEraseBkgnd 消息处理函数。这样就实现了窗口背景的重绘。

本章小结

本章主要介绍框架窗口的定制。通过本章的学习,读者应该达到以下几点:

- 掌握改变窗口效果的方法。
- 掌握增加闪屏效果的方法。
- 掌握添加窗口背景的方法。

[G e n e r a l I n f o r m a t i o n]
书名= V i s u a l C + + 图形用户界面开发指南
作者= 李博轩等编著
页数= 4 1 0
S S 号= 0
出版日期=

书名页
版权页
前言
目录
正文